O'REILLY®

Como ser um programador melhor

UM MANUAL PARA PROGRAMADORES QUE SE IMPORTAM COM CÓDIGO

novatec

Pete Goodliffe

Elogios a Como ser um programador melhor

Como ser um programador melhor está repleto de experiências e transmite a sabedoria acumulada em toda uma carreira nos negócios de software. Capítulos concisos, sobre um único assunto, tornam o livro realmente legível, com temas comuns abordados de todos os ângulos. Se você é um engenheiro de software procurando ir do bom para o ótimo, este livro é ideal para você. Vou usá-lo com os desenvolvedores iniciantes pelos quais sou responsável como orientador.

– *Andrew Burrows*
desenvolvedor líder

Goodliffe toma o assunto bem amplo da programação de computadores e consegue dividi-lo em uma narrativa clara, estimulante e cativante. Ele tem um talento particular para dizer coisas que parecem óbvias, mas que eu não havia percebido até ele dizê-las. Qualquer um que aspire ser um ótimo programador deve ler este livro.

– *Greg Law*
cofundador e CEO da Undo Software

Pete Goodliffe combina a teoria e a prática com sucesso. Quando algo deve ser feito de determinada maneira, ele não faz rodeios. Quando há áreas cinzentas, ele explica claramente os diferentes pontos de vista. Se considerar e aplicar o que ele diz, você irá se beneficiar e se aperfeiçoar; você se tornará um programador melhor. De modo geral, este livro está cheio de experiências destiladas do mundo real, misturadas com humor, para proporcionar sabedoria com serenidade.

– *Dr. Andrew Bennett*
Bacharel em engenharia/PhD/MIET/MIEEE

Este livro irá alimentar sua paixão pela arte e pela ciência da programação. Pete compreende que um ótimo software é o resultado de

boas pessoas fazendo o seu melhor trabalho. Ele mostra como fazer isso por meio de boas práticas de codificação, uma atitude adequada e bons relacionamentos, com muitos exemplos. E, como bônus, é uma leitura bem divertida!

– *Lisa Crispin*
coautora de *Agile Testing: A Practical Guide or Testers and Agile Teams*

Pete tem muita experiência como programador e orientador. Neste livro, ele dedicou a mesma atenção aos detalhes, classificando e descrevendo essas experiências, como ele faz na tarefa de ser verdadeiramente um programador. Conhecer programação é somente uma parte de "ser um programador" e, independentemente de a codificação ser uma novidade ou de você já ser experiente, ou se está começando a orientar alguém, este é um baú de tesouros cheio de conselhos sobre como lidar com isso – de alguém que realmente tem conhecimento. É um manual sobre os diversos obstáculos que você encontrará e como administrá-los de forma segura e eficiente.

– *Steve Love*
editor da revista *C Vu*

Com muita frequência, os programadores se dividem em programadores médios e desenvolvedores que são estrelas ou ninjas. Onde houver uma estrela, haverá uma base de código ruim, com classes que não funcionam e fluxos de controle desorganizados. Onde houver um ninja, haverá bugs misteriosos e problemas de compilação que surgem no meio da noite. Onde houver programadores médios, haverá distribuição. No longo prazo, em que ponto da distribuição uma pessoa está importa menos do que para onde ela está indo. Se quiser dividir os programadores em dois grupos, há programadores que irão melhorar e outros que não. Você deve se importar com o primeiro grupo. Este livro foi escrito para esses programadores.

– *Kevlin Henney*
consultor, palestrante e autor de *97 Things Every Programmer Should Know*

Este livro é bem chato e o peixe na capa não me convenceu.

– *Alice Goodliffe*

12 anos

ISBN: 978-85-7522-764-0

Histórico de edições impressas:

Agosto/2016 Primeira reimpressão
Janeiro/2015 Primeira edição

Para minha esposa Bryony, que eu amo.
E para nossas três filhas maravilhosas.
Salmo 150.

Sumário

Parte II ▪ A prática leva à perfeição

Também de Pete Goodliffe

Code Craft: The Practice of Writing Excellent Code
(No Starch Press)

97 Things Every Programmer Should Know
(O'Reilly, três capítulos de contribuição)

Beautiful Architecture
(O'Reilly, um capítulo de contribuição)

Introdução

Você se importa com código. É apaixonado por programação. É o tipo de desenvolvedor que gosta de criar softwares realmente bons. E você escolheu este livro porque quer fazer isso *melhor ainda*. Boa pedida.

Este livro irá ajudá-lo.

O objetivo é fazer exatamente o que está escrito na capa: ajudar você a se tornar um programador melhor. Mas o que isso quer dizer exatamente?

Muito cedo na carreira de qualquer programador, há uma percepção de que há mais para ser um ótimo codificador do que simplesmente entender a sintaxe e ter domínio sobre o design básico. Os programadores incríveis – aquelas pessoas produtivas, que criam códigos maravilhosos e trabalham de modo eficiente com outras pessoas – sabem muito mais. Há métodos de trabalho, atitudes, abordagens, técnicas e *idioms* aprendidos ao longo do tempo que aumentam a sua eficiência. Há habilidades sociais úteis e todo um conjunto de conhecimentos tribais a serem adquiridos.

E, é claro, você deve aprender a sintaxe e conhecer o design.

Esse é exatamente o assunto deste livro. A obra é um catálogo de técnicas e abordagens úteis para a arte da programação e ajudará você a se tornar um programador melhor.

Não vou fingir dizendo que este é um tratado completo. O campo é vasto. Há sempre mais para aprender, com novos territórios sendo descobertos a cada dia. Estes capítulos constituem simplesmente o fruto de mais de quinze anos de meu trabalho como programador profissional. Já vi muitos códigos e cometi muitos erros. Não vou alegar que sou um expert; apenas tenho bastante experiência. Se puder aprender com os erros que cometi e se inspirar com a minha experiência, você estará um passo à frente em sua carreira de desenvolvimento.

O que será discutido?

Os assuntos discutidos neste livro cobrem toda a gama da vida do desenvolvedor de software:

- preocupações no nível de código que afetam a maneira de escrever linhas individuais de código bem como o modo de fazer o design de seus módulos de software;
- técnicas práticas que ajudarão você a trabalhar melhor;
- demonstrações de atitudes e abordagens eficientes a serem adotadas, que ajudarão você a se tornar supereficiente e a ter uma base bem sólida;
- truques e dicas procedurais e organizacionais que ajudarão você a florescer enquanto estiver encarcerado na produção de software.

Não há nenhum preconceito contra qualquer linguagem ou mercado em particular neste livro.

Quem deve ler este livro?

Você!

Independentemente de ser um expert no mercado, um desenvolvedor experiente, um profissional iniciante ou um codificador por hobby, este livro será adequado para você.

Como ser um programador melhor tem como objetivo ajudar os programadores de qualquer nível a melhorar. É uma alegação importante, mas há sempre algo que podemos aprender e sempre há espaço para melhorias, não importa a sua experiência como programador. Cada capítulo oferece a oportunidade de rever suas habilidades e revela maneiras práticas de se aperfeiçoar.

O único pré-requisito para usar este livro é que você *queira* se tornar um programador melhor.

A estrutura

As informações contidas neste livro são apresentadas em uma série de capítulos simples e independentes, cada qual discutindo um único assunto. Se você for tradicional, poderá lê-los na sequência, do início ao fim. Contudo sinta-se à vontade para ler os capítulos na ordem que quiser. Vá direto para os capítulos que pareçam ser mais pertinentes, se isso o deixar mais feliz.

Os capítulos são apresentados em cinco partes:

você.escreve(código);

Iniciamos bem na base, diretamente no código – local em que os programadores se sentem mais à vontade. Essa seção apresenta técnicas importantes para escrita de código e mostra maneiras de escrever o melhor código possível. A escrita e a leitura de código, o design e os métodos para escrever um código robusto são discutidos.

A prática leva à perfeição

Afastando-se do código, essa parte discute as *práticas* importantes de programação que ajudam a fazer de você um programador melhor. Conheceremos atitudes e abordagens saudáveis relacionadas à tarefa de codificação e técnicas sólidas que ajudarão a criar um código melhor.

Envolvendo-se pessoalmente

Esses capítulos exploram em detalhes a maneira de atingir a excelência em sua vida pessoal de programação. Daremos uma olhada em como aprender com eficiência, considerar a maneira de nos comportarmos de forma ética, encontrar desafios estimulantes, evitar a estagnação e melhorar o bem-estar físico.

Conseguir que tudo seja feito

Esses capítulos discutem maneiras práticas de *conseguir que tudo seja feito*: disponibilizar o código no prazo, sem que haja desvios ou atrasos.

O que as pessoas buscam

O desenvolvimento de software é uma atividade social. Esses capítulos

mostram como trabalhar bem com os outros habitantes da fábrica de software.

Mais importante que a ordem com que você irá consumir esses capítulos é a maneira pela qual o material será abordado. Para realmente melhorar, você deverá aplicar o que for lido na prática. A estrutura de cada capítulo foi projetada para ajudá-lo nessa tarefa.

Em cada capítulo, o assunto em questão é detalhado em uma prosa fluente, com bastante clareza. Você irá rir, chorar e perguntar por quê. A conclusão de cada capítulo inclui as subseções a seguir:

Perguntas
: Uma série de perguntas que você deve considerar e responder. *Não pule* essas perguntas! Elas não pedem que você regurgite as informações que acabaram de ser lidas. As perguntas estão lá para fazer você pensar mais profundamente, ir além do material original e descobrir como o assunto se entrelaça à sua experiência atual.

Veja também
: Ligações para qualquer capítulo relacionado no livro, com uma explicação sobre como os capítulos se associam.

Tente isto...
: Por fim, cada capítulo é concluído com um desafio simples. É uma tarefa específica, que ajudará você a melhorar e a aplicar o assunto ao seu regime de codificação.

Ao longo de cada capítulo, há *pontos-chave* particularmente importantes. Eles estão em destaque para que você não deixe de vê-los.

> **PONTO-CHAVE** Este é um ponto-chave. Preste atenção.

À medida que você ler cada capítulo, por favor, passe um tempo considerando as perguntas e os desafios da seção *Tente isto...* Não os menospreze. Eles são uma parte importante de *Como ser um programador melhor*. Se você simplesmente passar os olhos pelas informações de cada capítulo, elas serão apenas isto: informações. Esperamos que elas sejam interessantes. Sem dúvida, serão informativas. Porém é pouco provável

que farão de você um programador muito melhor.

Você precisa ser desafiado e absorver o que foi lido para aumentar o seu conjunto de habilidades de programação. Esses exercícios finais não consumirão muito tempo. Honestamente. E eles ajudarão a consolidar o tema de cada capítulo em sua mente.

Uma nota para os orientadores

Este livro foi criado para servir como uma ferramenta valiosa para a orientação de colegas programadores. Você pode usá-lo em relações de um para um ou em um grupo de estudo.

A melhor maneira de usar este material é *não* trabalhar em conjunto em cada seção. Em vez disso, leiam um capítulo separadamente e então se reúnam para discutir o conteúdo. As perguntas realmente funcionam como um ponto de partida para a discussão, portanto é uma boa ideia começar por ali.

Como entrar em contato conosco

Envie seus comentários e questões sobre este livro à editora escrevendo para: *novatec@novatec.com.br*.

Temos uma página web para este livro, na qual incluímos erratas, exemplos e quaisquer outras informações adicionais.

- Página da edição em português

  http://www.novatec.com.br/catalogo/7522415-programador-melhor

- Página da edição original em inglês

  http://bit.ly/becoming_a_better_programmer

Para obter mais informações sobre os livros da Novatec Editora, acesse nosso site em: *http://www.novatec.com.br*.

Agradecimentos

Escrever um livro é uma empreitada surpreendentemente grande: uma

que tem a tendência de dominar a sua vida e engolir outras pessoas no turbilhão ao longo do caminho. Há muitas pessoas que, de algum modo, contribuíram para a publicação deste livro, desde as primeiras versões preliminares do material até ele se tornar o volume completo que está em sua estante.

Minha esposa maravilhosa Bryony me apoiou pacientemente (e me suportou) enquanto eu estava envolvido nessa tarefa, além de em várias outras tarefas com as quais estava lidando. Eu te amo e tenho enorme consideração por você. Alice e Amelia me proporcionaram muitas distrações bem-vindas: vocês tornam a vida divertida!

Algumas partes deste livro foram baseadas em artigos que escrevi durante os últimos anos. Steve Love, o querido editor da revista *C Vu* da ACCU, contribuiu com feedbacks valiosos sobre muitos desses artigos, e seu incentivo e suas opiniões sábias sempre foram apreciados. [Se você não conhece a ACCU (*http://www.accu.org/*), ela é uma organização incrível para programadores que se importam com código.]

Muitos amigos e colegas contribuíram com inspiração, críticas e feedbacks valiosos. Essas pessoas incluem minha família Akai: Dave English, Will Augar, Łukasz Kozakiewicz e Geoff Smith. Lisa Crispin e Jon Moore contribuíram com ideias do ponto de vista de QA (Quality Assurance, ou Controle de qualidade); Greg Law me ensinou alguns fatos sobre bugs enquanto Seb Rose e Chris Oldwood fizeram revisões muito apreciadas e nos momentos certos.

Os revisores técnicos – Kevlin Henney, Richard Warburton e Jim Brikman – ofereceram muitos feedbacks valiosos e ajudaram a dar forma ao texto que você está lendo. Sou grato a eles pelas contribuições inteligentes.

A excelente equipe de editores e de gênios da produção da O'Reilly trabalhou arduamente neste livro, e sou grato pela sua cuidadosa atenção. Em particular, o trabalho inicial de Mike Loukides e de Brian MacDonald ajudou consideravelmente a dar forma a esse material.

Lorna Ridley trabalhou muito para garantir a qualidade deste livro.

CAPÍTULO 1

Importar-se com o código

Das pessoas com quem nos importamos vem a coragem.[1]

– Lao-Tsé

Não é preciso ser Sherlock Holmes para descobrir que bons programadores escrevem bons códigos. Programadores ruins... não. Esses produzem monstruosidades do tamanho de um elefante, que o restante de nós deve limpar. Você quer escrever um código bom, certo? Você quer ser um bom programador.

Um bom código não surge simplesmente do nada. Não é algo que acontece por sorte quando os planetas se alinham. Para ter um bom código, é preciso trabalhar nele. Arduamente. E você só terá um código bom se realmente se *importar* com códigos bons.

> **PONTO-CHAVE** Para escrever um bom código, você deve se *importar* com ele. Para ser um programador melhor, é necessário investir tempo e esforço.

Uma boa programação não nasce somente da competência técnica. Já vi programadores altamente intelectuais que conseguem produzir algoritmos intensos e impressionantes, conhecem o padrão de sua linguagem de cor, mas escrevem o pior código possível. É um código difícil de ler, de usar e de modificar. Já vi programadores mais humildes que se atêm a um código bem simples, porém escrevem programas elegantes e expressivos com os quais é uma satisfação trabalhar.

Com base em meus anos de experiência na produção de software, concluí que a verdadeira diferença entre programadores medíocres e programadores excelentes é esta: *atitude*. Uma boa programação resulta da adoção de uma abordagem profissional e de *querer* escrever o melhor software que você puder, levando em conta as limitações e as pressões do

mundo real na produção de software.

O código para o inferno está cheio de boas intenções. Para ser um programador excelente, você deverá estar acima das boas intenções e realmente se *importar* com o código – adote perspectivas positivas e desenvolva atitudes saudáveis. Bons códigos são cuidadosamente trabalhados por mestres-artesãos, e não implementados por programadores desleixados, sem muito planejamento, ou criados misteriosamente por aqueles que se autodenominam gurus da codificação.

Você quer escrever bons códigos. Quer ser um bom programador. Portanto você se importa com o código. Isso significa que você deve agir de acordo com esta postura, por exemplo:

- Em qualquer situação de codificação, você se recusa a criar um hack que somente *pareça* funcionar. Você se esforça para compor um código elegante, que esteja claramente correto (e tem bons testes para mostrar que ele está correto).
- Você escreve um código que *revele a intenção* (que outros programadores possam entender facilmente), que *possa ser mantido* (que você ou outros programadores serão capazes de modificar facilmente no futuro) e que esteja *correto* (você executa todos os passos possíveis para determinar que o problema tenha sido *resolvido*, em vez de simplesmente fazer parecer que o programa funciona).
- Você trabalha bem com outros programadores. Nenhum programador é uma ilha. Poucos programadores trabalham sozinhos; a maioria trabalha em uma equipe de programadores, seja em um ambiente de empresa ou em um projeto de código aberto. Você leva os outros programadores em consideração e cria um código que os demais possam ler. Você quer que a equipe escreva o melhor software possível em vez de querer parecer ser o mais inteligente.
- Sempre que mexer em um código, você se esforça para torná-lo melhor do que estava (mais estruturado, mais bem testado e mais compreensível...).
- Você se importa com código e com programação, portanto está

constantemente aprendendo novas linguagens, idioms e técnicas. Porém você somente os aplica quando for apropriado.

Felizmente, você está lendo este livro porque *realmente* se importa com código. Está interessado por ele. O código é a sua paixão. Você gosta de criá-lo de forma adequada. Continue lendo e veremos como transformar essa preocupação com o código em ações práticas.

À medida que fizer isso, nunca se esqueça de se divertir com a programação. Aprecie remover códigos para solucionar problemas intrincados. Crie softwares que o deixem orgulhoso.

PONTO-CHAVE Não há nada de errado em ter uma resposta emocional a um código. Ter orgulho de seu bom trabalho ou ficar aborrecido com um código ruim é saudável.

Perguntas

1. Você se *importa* com código? Como isso se manifesta no trabalho que você produz?
2. Você quer se aperfeiçoar como programador? Em quais áreas você acha que deve trabalhar mais?
3. Se você não se importa com código, por que está lendo este livro?!
4. O quão exata é a afirmação *Bons programadores escrevem código bom. Programadores ruins... não?* É possível que bons programadores escrevam código ruim? Como?

Veja também

- *Desenvolvimento de software é...* (Capítulo 14) – Com *o que* é que nós nos importamos?
- *Fale!* (Capítulo 36) – Nós nos importamos em trabalhar com um código bom. Também devemos nos importar em trabalhar com boas *pessoas*.

TENTE ISTO... Comprometa-se agora com a melhoria de suas habilidades de programação. Envolva-se com o que você ler neste livro, responda às perguntas e procure realizar todos os desafios da seção *Tente isto...*

1 N.T.: Essa e todas as demais citações contidas neste livro são traduções livres, feitas de acordo com a citação original em inglês.

PARTE I

você.escreve(código);

A primeira parte deste livro trata da vida nas linhas de frente: nossa batalha diária com o código.

Daremos uma olhada nos detalhes de baixo nível com que os programadores se deleitam: como escrever linhas de código individuais, como melhorar seções do código e como planejar um caminho por um código existente. Também investiremos tempo nos preparando para o inesperado: lidar com erros, escrever um código robusto e a obscura arte de identificar bugs. Por fim, daremos uma olhada no quadro geral: consideraremos os aspectos de design de nossos sistemas de software e investigaremos as consequências técnicas e práticas desses designs.

CAPÍTULO 2
Mantendo as aparências

As aparências enganam.

– Esopo

Ninguém gosta de trabalhar com código confuso. Ninguém quer entrar em um lamaçal de formatação irregular e inconsistente nem lutar contra nomes sem nexo. Não é divertido. Não é produtivo. É o purgatório do programador.

Nós nos importamos com código. E, naturalmente, nos importamos com a estética do código; é o que de imediato determina o nível de facilidade que teremos para trabalhar com uma seção do código. Praticamente todo livro sobre programação tem um capítulo sobre apresentação. Ah, veja, este livro também tem um. Vá entender.

Infelizmente, os programadores se preocupam tanto com a apresentação do código que acabam brigando por causa dela. É desse tipo de assunto que surgem as guerras santas. Disso e de qual é o melhor editor.[1] Tabulações *versus* espaços. Posicionamento de chaves. Colunas por linha. Uso de letras maiúsculas. Eu tenho minhas preferências. Você tem as suas.

A *lei de Godwin* afirma que à medida que uma discussão na Internet se estende, a probabilidade de uma comparação com os nazistas ou com Hitler se aproxima de um. *A lei de Goodliffe (revelada aqui) afirma que à medida que qualquer discussão sobre* layout de código aumenta, a probabilidade de ela se tornar uma discussão infrutífera se aproxima de um.

Bons programadores se importam muito com uma boa apresentação do código. Porém eles estão acima desse tipo de disputa mesquinha. Vamos agir como adultos.

PONTO-CHAVE Pare de brigar por causa de layout de código. Adote uma atitude saudável em relação à sua apresentação de código.

Nosso foco míope em layout é ilustrado claramente pela clássica revisão de código que não é funcional. Dada uma seção de código, a tendência é identificar uma infinidade de falhas na apresentação. (Especialmente se você fizer somente uma leitura superficial, o layout é tudo com que você implicará.) Você achará que fez vários comentários úteis. As falhas de design serão totalmente ignoradas porque a posição de uma chave está incorreta. Realmente, parece que quanto mais ampla for a revisão de código e quanto mais rápida ela for feita, maior será a chance de essa cegueira atacar.

A apresentação é eficaz

Não podemos fingir que a formatação do código não seja importante. Mas saiba por quê. Uma boa formatação de código *não* é aquela que você acha ser a mais bonita. Não organizamos o código para pôr em prática nossos profundos conhecimentos artísticos. (Você pode ouvir os críticos de arte da codificação? *Queriiiiiido, veja essa maravilhosa moldura Pré-Rafaelita nessa instrução switch aninhada*. Ou: *você deve apreciar o subtexto profundo que há nesse método*. Acho que não.)

Um bom código é claro. É consistente. O layout é quase invisível. Uma boa apresentação não chama a atenção nem provoca distrações; ela serve somente para revelar o propósito do código. Isso ajuda os programadores a trabalhar com o código de forma eficiente. Reduz o esforço necessário para manter o código.

PONTO-CHAVE Uma boa apresentação do código revela o seu propósito. Não é um empreendimento artístico.

As técnicas para uma boa apresentação são importantes não pela beleza, mas para *evitar erros* em seu código. Como exemplo, considere o seguinte trecho de código C:

```
bool ok = thisCouldGoWrong();
if (!ok)
   fprintf(stderr, "Error: exiting...\n");
```

```
    exit(0);
```

Você pode perceber a intenção do autor nesse caso: `exit(0)` deveria ser chamado somente se o teste falhasse. No entanto a apresentação ocultou o verdadeiro comportamento: o código sempre executará `exit`. As opções de layout tornaram o código suscetível a erros.[2]

Os nomes têm um efeito profundo semelhante. Nomes ruins podem fazer mais do que simplesmente distrair: podem ser muito perigosos. Qual desses nomes é ruim?

```
bool numberOfGreenWidgets;
string name;
void turnGreen();
```

`numberOfGreenWidgets` é uma variável, certo? É evidente que um contador não é representado por um tipo booleano. Não, é uma pergunta capciosa. Todos os nomes são ruins. A string não armazena realmente um nome, mas o nome de uma cor; ela é definida pela função `turnGreen()`. Portanto esse nome de variável engana. E `turnGreen` foi implementada desta maneira:

```
void turnGreen()
{
   name = "yellow";
}
```

Todos os nomes enganam!

Esse é um exemplo artificial? Talvez; no entanto, depois de uma manutenção um pouco descuidada, o código poderá rapidamente chegar a esse estado. O que acontece quando trabalhamos com um código como esse? Bugs. Muitos e muitos bugs.

> **PONTO-CHAVE** Precisamos de uma boa apresentação para evitar erros no código, e não para criar uma arte ASCII bonita.

Deparar-se com um layout inconsistente e com nomes confusos é um sinal evidente de que a qualidade do código não é muito boa. Se os autores não tiveram cuidado com o layout, provavelmente não tiveram nenhuma preocupação com outras questões vitais relacionadas à qualidade (por exemplo, um bom design, testes abrangentes etc.).

É uma questão de comunicação

Escrevemos código para dois públicos-alvo. O primeiro é o compilador (ou o runtime da linguagem). Essa fera estará totalmente satisfeita em ler qualquer código antigo ruim e o transformará em um programa executável da única maneira que ele souber. Isso será feito de forma impassível, sem nenhum julgamento sobre a qualidade do que você lhe fornecer nem sobre o estilo com o qual o código for apresentado. É mais um exercício de conversão do que qualquer tipo de "leitura" de código.

O outro público-alvo, mais importante, são os *outros programadores*. Escrevemos código para ser executado por um computador, porém ele será *lido* por seres humanos. Esses incluem:

- Você, nesse exato momento em que está escrevendo o código. O código deve estar claro como água para que você não cometa erros de implementação.
- Você, algumas semanas (ou meses) depois, quando preparar o software para que uma versão seja disponibilizada.
- Outras pessoas de sua equipe que farão a integração de seus trabalhos com esse código.
- O programador responsável pela manutenção (que poderá ser você ou outro programador) anos depois, quando estiver investigando um bug em uma versão antiga.

Um código difícil de ser lido é um código com o qual é difícil trabalhar. É por isso que nos esforçamos para ter uma apresentação clara, agradável e útil.

> **PONTO-CHAVE** Lembre-se das pessoas para quem você está escrevendo o código: os outros.

Já vimos que o código pode parecer bonito, porém ser obscuro quanto ao seu propósito. Ele também pode parecer bonito, mas ser excessivamente difícil de ser mantido. Um ótimo exemplo desse caso está na "caixa de comentário". Alguns programadores gostam de apresentar banners de comentários em belas caixas ASCII que são verdadeiras obras de arte:

```
/*************************************************************************
* Este é um comentário elegante. *
* Observe que há asteriscos do *
* lado direito da caixa. Uau, parece organizado. *
* Espero que eu jamais tenha de corrigir esse erro de digiitação. *
*************************************************************************/
```

É bonitinho, mas não é fácil de ser mantido. Se quiser alterar o texto do comentário, você deverá trabalhar manualmente nos marcadores de comentário à direita das linhas. Francamente, esse é um estilo de apresentação sádico, e as pessoas que o escolheram não valorizam o tempo e a sanidade de seus colegas. (Ou elas esperam tornar a edição desses comentários tão incrivelmente maçante que ninguém ousará corrigir sua prosa.)

Layout

Se alguém quiser escrever com um estilo claro, deverá ter um raciocínio claro antes de tudo.

– Johann von Goethe

As preocupações com o layout do código incluem indentação, uso de espaços em branco em torno dos operadores, uso de letras maiúsculas, posicionamento de chaves (seja com o estilo K&R, Allman, Whitesmith ou algo do tipo) e o velho debate da indentação com tabulações *versus* espaços. Em cada uma dessas áreas, há várias decisões de layout que podem ser tomadas, e cada opção tem bons motivos para ser recomendada. Desde que suas opções de layout melhorem a estrutura de seu código e ajudem a revelar o seu propósito, elas serão boas.

Uma olhada rápida em seu código deverá revelar o seu formato e a estrutura. Em vez de discutir sobre posicionamento de chaves, há considerações mais importantes sobre o layout que iremos explorar nas próximas seções.

Crie uma boa estrutura

Escreva o seu código como se estivesse em prosa.

Separe-o em capítulos, parágrafos e sentenças. Reúna os itens semelhantes

e separe o que for diferente. As funções são como capítulos. Em cada capítulo pode haver partes do código distintas, porém relacionadas. Separe-as em parágrafos inserindo linhas em branco entre elas. Não insira linhas em branco, a menos que haja uma quebra natural de "parágrafo". Essa técnica ajuda a dar ênfase ao fluxo e à estrutura.

Por exemplo:

```
void exampleFunction(int param)
{
   // Agrupamos o que está relacionado aos dados de entrada
   param = sanitiseParamValue(param);
   doSomethingWithParam(param);

   // Outras tarefas são inseridas em um "parágrafo" separado
   updateInternalInvariants();
   notifyOthersOfChange();
}
```

A ordem em que o código é apresentado é importante. Leve o leitor em consideração: coloque as informações mais importante antes, e não no final. Garanta que as APIs sejam lidas em uma ordem sensata. Coloque aquilo que for importante para o leitor no início de sua definição de classe. Isso significa que toda informação pública deverá vir antes das informações privadas. A criação de um objeto deve vir antes de seu uso.

Esse agrupamento pode ser expresso em uma declaração de classe como esta:

```
class Example
{
public:
   Example(); // gerenciamento do ciclo de vida antes
   ~Example();

   void doMostImportantThing(); // isso inicia um novo "parágrafo"
   void doSomethingRelated(); // cada linha aqui é como de fosse uma sentença

   void somethingDifferent(); // esse é outro parágrafo
   void aRelatedThing();

private:
   int privateStuffComesLast;
};
```

Prefira escrever blocos menores de código. Não crie uma função com

cinco "parágrafos". Considere separá-la em cinco funções, cada qual com um nome bem definido.

Consistência

Evite a preciosidade em relação aos estilos de layout. Escolha um. Use-o de forma consistente. É melhor ser idiomático – use aquele que melhor se enquadre a sua linguagem. Siga o estilo das bibliotecas-padrão.

Escreva códigos usando as mesmas convenções de layout usadas pelo restante de sua equipe. Não utilize seu próprio estilo só porque você acha que ele é mais bonito ou melhor. Se não houver consistência em seu projeto, considere a adoção de um *padrão de codificação* ou de um *manual de estilo*. O documento não precisa ser extenso ou draconiano; um conjunto de princípios de layout em relação ao qual a equipe tenha concordado será suficiente. Nessa situação, deve haver um consenso mútuo sobre os padrões de codificação; esses não devem ser impostos.

Se você estiver trabalhando em um arquivo que não siga as convenções de layout do restante de seu projeto, use as convenções de layout nesse arquivo.

Garanta que os IDEs e os editores de código-fonte de toda a equipe estejam configurados da mesma maneira. Uniformize o tamanho das tabulações. Defina a posição das chaves e as opções para layout de comentários de forma idêntica. Faça com que as opções de fim de linha sejam iguais. Isso é particularmente importante em projetos para diversas plataformas, em que ambientes de desenvolvimento bem diferentes são usados simultaneamente. Se você não for zeloso quanto a isso, o código-fonte naturalmente será frágil e inconsistente; você criará um código ruim.

História de guerra: as guerras sobre os espaços em branco

Juntei-me a um projeto em que os programadores não prestavam nenhuma atenção à apresentação. O código era confuso, inconsistente e desagradável. Pedi para que um padrão de codificação fosse introduzido.

Todos os desenvolvedores concordaram que essa era uma boa ideia e estavam

dispostos a concordar com as convenções para a nomenclatura, o layout e a hierarquia de diretórios. Foi um enorme passo à frente. O código começou a crescer de forma mais organizada.

No entanto havia uma questão sobre a qual simplesmente não conseguíamos chegar a um consenso. Você adivinhou: *tabulações* ou *espaços*. Quase todos preferiam usar indentação com quatro espaços. Um indivíduo jurava que as tabulações eram melhores. Ele discutia, reclamava e se recusava a mudar o seu estilo de codificação. (Provavelmente, ele deve continuar discutindo a questão até os dias de hoje.)

Como havíamos feito algumas melhorias significativas, e para evitar discussões desnecessárias que provocassem uma divisão, deixamos a questão de lado. Todos nós usávamos espaços em branco. Ele usava tabulações.

Como resultado, trabalhar com o código continuou sendo difícil e frustrante. Editar o código era uma tarefa surpreendentemente inconsistente; às vezes, seu cursor movia um espaço por vez, às vezes, ele dava um salto. Algumas ferramentas exibiam o código de forma razoavelmente boa se você definisse uma parada de tabulação apropriada. Outras (incluindo o visualizador de nosso sistema de controle de versões e o sistema online de revisão de código) não podiam ser ajustadas e exibiam um código irregular, difícil de olhar.

Nomes

Quando uso uma palavra, disse Humpty Dumpty em um tom de desdém,ela quer dizer exatamente o que eu quero que ela signifique – nem mais e nem menos.

– Lewis Carroll

Damos nomes a vários itens: variáveis, funções e métodos, tipos (por exemplo, enumerações, classes), namespaces e pacotes. Igualmente importantes são os itens maiores como arquivos, projetos e programas. As APIs públicas (por exemplo, interfaces de biblioteca ou APIs de web services) talvez sejam os itens mais significativos aos quais damos nomes, pois as APIs públicas “disponibilizadas” em uma versão, com muita frequência, são escritas a ferro e fogo e são particularmente difíceis de mudar.

Um nome representa a identidade de um objeto; ele descreve o item, indica o seu comportamento e o uso pretendido. Uma variável indevidamente nomeada pode ser *muito* confusa. Um bom nome é descritivo, correto e idiomático.

Você só pode nomear algo se souber *exatamente* o que é que está sendo nomeado. Se você não conseguir descrever um item claramente ou não souber para que ele será usado, simplesmente não será possível lhe dar um bom nome.

Evite redundância

Ao dar nomes, evite redundância e explore o contexto. Considere este código:

```
class WidgetList {
   public int numberOfWidgets() { ... }
};
```

O nome do método `numberOfWidgets` é desnecessariamente longo e repete a palavra *Widget*. Isso torna o código difícil e maçante para ler. Como esse método retorna o tamanho da lista, ele poderá simplesmente ser chamado de `size()`. Não haverá confusão, pois o contexto da classe que inclui o método claramente define o significado de *size* nesse caso.

Evite palavras redundantes.

Certa vez, trabalhei em um projeto com uma classe chamada `DataObject`. Era uma obra-prima de nomenclatura desconcertante e redundante.

Seja claro

Favoreça a clareza em relação à concisão. Os nomes não precisam ser curtos para economizar na digitação – você *lerá* o nome da variável muito mais vezes do que irá digitá-la. Entretanto há um caso de uso de nomes de variável com uma única letra: variáveis de contador em loops curtos tendem a ser lidas de modo mais claro. Novamente, o contexto é importante!

Os nomes não precisam ser enigmáticos. O garoto-propaganda desse caso é a notação húngara[3]. Ela não é útil.

Acrônimos rebuscados ou jogos de palavra “divertidos” não ajudam.

Seja idiomático

Prefira nomes idiomáticos. Empregue as convenções para uso de letras maiúsculas usadas com mais frequência em sua linguagem. São convenções eficazes que você deve deixar de seguir somente se houver um bom motivo. Por exemplo:

- Em C, as macros normalmente recebem nomes com letras maiúsculas.
- Nomes iniciados com letras maiúsculas geralmente denotam tipos (por exemplo, uma classe), enquanto nomes com letras minúsculas são reservados para métodos e variáveis. Esse pode ser um idiom tão universalmente aceito que não segui-lo poderá resultar em um código confuso.

Seja preciso

Certifique-se de que seus nomes sejam precisos. Não chame um tipo de `WidgetSet` se ele se comportar como um array de widgets. Um nome impreciso pode fazer com que o leitor faça suposições inválidas sobre o comportamento ou as características do tipo.

Torne-se apresentável

Nós nos deparamos com código formatado de maneira inadequada o tempo todo. Tome cuidado ao trabalhar com esse tipo de código.

Se for preciso “dar uma organizada” no código, jamais altere a apresentação ao mesmo tempo que fizer mudanças funcionais. Faça check-in das mudanças relacionadas à apresentação no sistema de controle de versões como um passo separado. *Em seguida*, altere o comportamento do código. É confuso ver commits que misturem os dois passos. As mudanças de layout podem mascarar erros na funcionalidade.

> **PONTO-CHAVE** Jamais altere a apresentação e o comportamento ao mesmo tempo. Faça com que sejam alterações separadas no sistema de controle de versões.

Não pense que você deve escolher um estilo de layout e permanecer fielmente com ele pelo resto da vida. Reúna feedback continuamente sobre o modo como as opções de layout afetam a maneira de trabalhar com o código. Aprenda com o código que você ler. Adapte o seu estilo de apresentação à medida que ganhar experiência.

Ao longo de minha carreira, mudei lentamente o meu estilo de codificação em direção a um layout mais consistente e que permitisse modificações mais facilmente.

De tempos em tempos, todo projeto considera a execução de ferramentas automatizadas de layout pela árvore de códigos-fonte ou a adição delas como um hook pré-commit. Sempre vale a pena investigar essas ferramentas, mas raramente vale a pena usá-las. Essas ferramentas de layout tendem a ser (compreensivelmente) simplistas e jamais poderão lidar com as sutilezas da estrutura do código no mundo real.

Conclusão

Pare de brigar por causa da apresentação do código. Favoreça a adoção de uma convenção comum em seu projeto, mesmo que não seja o estilo de layout de sua preferência.

Contudo tenha uma opinião sólida sobre o que constitui um bom estilo de layout e por quê. Aprenda continuamente e adquira mais experiência ao ler o código de outras pessoas.

Esforce-se para ter consistência e clareza no layout de seu código.

Perguntas

1. Você deve alterar o layout de códigos legados para atender aos padrões de codificação da empresa? Ou é melhor manter o estilo original do autor? Por quê?
2. Qual é o valor das ferramentas de reformatação de código? O quanto isso depende da linguagem que você está usando?
3. O que é mais importante: uma boa apresentação do código ou um bom design?

4. Qual é o nível de consistência do código de seu projeto atual? Como isso pode ser melhorado?
5. Tabulações ou espaços? Por quê? Isso importa?
6. É importante seguir o layout e as convenções de nomenclatura de uma linguagem? Ou é mais produtivo adotar um "estilo doméstico" diferente para que você possa diferenciar o código de sua aplicação do código da biblioteca-padrão?
7. O uso de editores de código que deem destaque à sintaxe usando cores significa que haverá menos requisitos relacionados a determinadas preocupações com a apresentação, dado que as cores ajudam a mostrar a estrutura do código?

Veja também

- *Fale!* (Capítulo 36) – A escrita e a apresentação do código têm tudo a ver com comunicação. Esse capítulo discute como um programador se comunica, tanto no código quanto na escrita e na fala.
- *O fantasma de um código do passado* (Capítulo 5) – Discute como o seu estilo de programação se desenvolve ao longo do tempo. O estilo de apresentação do código é algo que provavelmente sofrerá adaptações à medida que você ganhar experiência.

TENTE ISTO... Analise as suas preferências de layout. Elas são idiomáticas, simples, claras e consistentes? Como isso pode ser melhorado? Você discorda de seus colegas de equipe sobre a apresentação? Como essas diferenças podem ser resolvidas?

---

1 É o vim e ponto final.

2 Esse não é apenas um exemplo acadêmico para preencher livros! Bugs sérios na vida real surgem desses tipos de erro. A vergonhosa vulnerabilidade de segurança *goto fail* de 2014 da Apple em sua implementação de SSL/TLS foi causada exatamente por esse tipo de erro de layout.

3 N.T.: Notação em que caracteres indicando os tipos das variáveis são usados como prefixos nos nomes.

CAPÍTULO 3
Escreva menos código!

Um mínimo bem usado é suficiente para tudo.

– Júlio Verne
A volta ao mundo em oitenta dias

É triste, mas é verdade: em nosso mundo moderno, simplesmente há código demais.

Posso lidar com o fato de que o motor do meu carro seja controlado por um computador. Obviamente, há software cozinhando a comida em meu forno de micro-ondas. E eu não ficaria surpreso se meus pepinos geneticamente modificados tivessem um microcontrolador embutido. Tudo bem, não é com isso que estou obcecado. Estou preocupado com todo o código *desnecessário* que existe por aí.

Simplesmente há muito código desnecessário em ação no mundo. Como ervas daninhas, essas linhas de código perversas entopem nossos preciosos bytes de armazenamento, ofuscam nossos históricos de controle de revisões, atrapalham obstinadamente o nosso desenvolvimento e utilizam um espaço precioso de código, sufocando o bom código em torno delas.

Por que há tanto código desnecessário?

Algumas pessoas gostam do som de sua própria voz. Você já conheceu pessoas assim; simplesmente não é possível fazê-las se calar. É o tipo de pessoa com quem você não quer ficar em festas. *Blá blá blá*. Outras pessoas gostam demais de seu próprio código. Gostam tanto que escrevem uma quantidade enorme dele: `{ bla->bla.bla(); }`.

Ou quem sabe sejam programadores com gerentes equivocados que julgam o progresso com base em milhares de linhas de código escritas em

um dia.

Escrever bastante código *não* significa que você criou bastante software. Na verdade, alguns códigos podem afetar negativamente a quantidade de software que você tem – eles ficarão no caminho, provocarão falhas e reduzirão a qualidade da experiência do usuário. É o equivalente à antimatéria na programação.

PONTO-CHAVE Menos código *pode* significar mais software.

Alguns de meus melhores trabalhos de melhoria de software corresponderam a remoções de código. Lembro-me com alegria de certa ocasião em que removi milhares de linhas de código de um sistema que estava se espalhando e as substituí por meras dez linhas de código. Que sensação maravilhosa de orgulho e de satisfação! Sugiro que você tente fazer isso algum dia.

Por que devemos nos importar?

Por que esse fenômeno é *ruim*, e não apenas irritante?

Há muitas razões para um código desnecessário ser a causa de todo mal. A seguir, temos algumas delas listadas:

- Escrever uma nova linha de código representa o nascimento de uma pequena forma de vida. Ela deverá ser alimentada com amor e se transformar em um membro útil e produtivo da sociedade de software antes de você poder disponibilizar um produto que a utilize.

  Ao longo da vida de seu sistema de software, essa linha de código deverá ser mantida. Cada linha de código tem um pequeno custo. Quanto mais código você escrever, mais alto será o custo. Quanto mais tempo uma linha de código viver, mais alto será o custo. É evidente que um código desnecessário deve morrer a tempo, antes que ele nos leve à falência.
- Mais código significa mais para ler e para entender – isso faz com que nossos programas sejam mais difíceis de ser compreendidos. Um código desnecessário pode mascarar o propósito de uma função ou

ocultar diferenças pequenas, porém importantes, em códigos semelhantes.

- Quanto mais código houver, mais trabalho será necessário para fazer modificações – será mais difícil alterar o programa.
- O código abriga bugs. Quanto mais código você tiver, mais lugares haverá para os bugs se esconderem.
- Um código duplicado é particularmente pernicioso: você pode corrigir um bug em uma cópia do código e, sem que você saiba, poderá ter outros 32 pequenos bugs idênticos em ação por aí.

Um código desnecessário é nefasto. Ele tem vários disfarces: componentes não utilizados, código morto, comentários sem propósito, verbosidade desnecessária e assim por diante. Vamos dar uma olhada em alguns desses em detalhes.

Lógica oscilante

Um tipo simples e comum de código inútil está no uso desnecessário de instruções condicionais e de construções lógicas tautológicas. Uma lógica oscilante é sinal de uma mente oscilante. Ou, no mínimo, de uma compreensão pobre das construções lógicas. Por exemplo:

```
if (expression)
   return true;
else
   return false;
```

pode ser escrito de forma mais simples e direta como

```
return expression;
```

Esse código não só é mais compacto como também é mais fácil de ser lido e, portanto, mais fácil de ser compreendido. Ele se parece mais com uma sentença em inglês, o que ajuda bastante os leitores humanos. E quer saber mais? O compilador não se importa nem um pouco com isso.

De modo semelhante, a expressão prolixa

```
if (something == true)
{
   // ...
```

```
}
```

será muito mais facilmente lida como

```
if (something)
```

Esses exemplos obviamente são simplistas. No mundo lá fora, vemos construções muito mais elaboradas; jamais subestime a capacidade de um programador complicar algo simples. Os códigos do mundo real estão repletos de construções como

```
bool should_we_pick_bananas()
{
   if (gorilla_is_hungry())
   {
      if (bananas_are_ripe())
      {
         return true;
      }
      else
      {
         return false;
      }
   }
   else
   {
      return false;
   }
}
```

que podem muito bem ser reduzidas a uma única linha:

```
return gorilla_is_hungry() && bananas_are_ripe();
```

Vá direto ao ponto e diga tudo claramente, porém de forma sucinta. Não se sinta envergonhado por conhecer o funcionamento de sua linguagem. Não é algo sujo e não crescerão pelos em suas mãos. Saber explorar a ordem pela qual as expressões são avaliadas fará você evitar muita lógica desnecessária em expressões condicionais. Por exemplo:

```
if ( a
   || (!a && b) )
{
   // que expressão complicada!
}
```

pode ser simplesmente escrito como

```
if (a || b)
{
   // assim não está melhor?
   // nem doeu, não é mesmo?
}
```

PONTO-CHAVE Expresse o código de forma clara e sucinta. Evite instruções desnecessariamente longas.

Refatoração

O termo *refatorar* entrou no léxico do programador nos anos 90. Essa palavra descreve um tipo particular de modificação de software e foi popularizada pelo livro *Refatoração: aperfeiçoando o projeto de código existente* de Martin Fowler.*

O termo, segundo a minha experiência, é indevidamente usado com frequência.

Ele descreve especificamente uma mudança feita na estrutura de um código existente (ou seja, em sua *fatoração*), que *não muda* o comportamento exibido. É essa última parte que normalmente é esquecida. Uma refatoração *somente* será uma refatoração se houver uma transformação do código-fonte em que o comportamento seja preservado. Uma "melhoria" que altere o modo como o programa reage (independentemente do nível de sutileza) não será uma refatoração; será uma melhoria. Uma "reorganização" que ajuste a UI não será uma refatoração; será uma reorganização.

Refatoramos para tornar o código mais legível, melhorar a estrutura interna, facilitar a manutenção do código e – com mais frequência – preparar o código para o acréscimo de funcionalidades no futuro.

Há conjuntos de refatorações simples que podem ser aplicadas em sequência no código. Muitos IDEs de linguagens oferecem suporte automatizado para isso. Essas transformações incluem: *Extract Class* (Extrair classe) e *Extract Method* (Extrair método), que separam uma funcionalidade em partes lógicas mais adequadas, e *Rename Method* (Renomear método) e *Pull Up/Pull Down* (Mover para cima/para baixo), que ajudam a mover o código para o lugar certo.

Uma refatoração adequada exige disciplina e é intensamente simplificada por uma boa suíte de testes de unidade que cubra o código em questão. Isso ajuda a provar que qualquer transformação realmente preservou o comportamento.

* Martin Fowler, *Refactoring: Improving the Design of Existing Code* (Boston: Addison-Wesley, 1999).

Duplicação

Uma duplicação desnecessária de código é perversa. Vimos esse crime ser perpetrado principalmente por meio da aplicação de *copiar e colar* na programação: ocorre quando um programador preguiçoso opta por não fatorar seções de código repetidas em uma função comum, porém as copia fisicamente de um local para outro em seu editor. Trabalho sujo. Esse pecado se agrava quando o código é colado com pequenas mudanças.

Quando um código é duplicado, você oculta a estrutura repetida e todos os bugs existentes são copiados. Mesmo que você corrija uma instância do código, haverá uma fila de bugs idênticos, prontos para atacar outro dia. Refatore seções de código duplicadas em uma única função. Se houver seções semelhantes de código com pequenas diferenças, capture essas diferenças em uma função com um parâmetro de configuração.

PONTO-CHAVE Não copie seções de código. Fatore-as em uma função comum. Utilize parâmetros para expressar qualquer diferença.

Isso é comumente conhecido como princípio *DRY*: *Don't Repeat Yourself!* (Não se repita). Nosso objetivo é ter um código "DRY", sem redundância desnecessária. No entanto saiba que fatorar um código semelhante em uma função compartilhada introduz um alto acoplamento entre essas seções de código. Ambos os códigos agora dependem de uma interface compartilhada; se ela for alterada, ambas as seções de código deverão ser ajustadas. Em várias situações, isso é perfeitamente apropriado; porém nem sempre é um resultado desejável, e pode provocar mais problemas no longo prazo do que a duplicação – portanto use o princípio DRY em seu código de forma responsável!

Nem toda duplicação de código é maliciosa ou é uma falha de programadores preguiçosos. A duplicação pode ocorrer por acidente também, por alguém reinventando a roda que essa pessoa não sabia que

existia. Ou pode ocorrer pela criação de uma nova função quando uma biblioteca de terceiros perfeitamente aceitável já existe. Isso é ruim porque é muito mais provável que a biblioteca existente esteja correta e já tenha sido depurada. Usar bibliotecas comuns faz você economizar esforço e o protege contra um mundo de falhas em potencial.

Há também padrões de duplicação no nível micro do código. Por exemplo:

```
if (foo) do something();
if (foo) do_something_else()
if (foo) do_more();
```

podem ser organizadamente encapsuladas em uma única instrução `if`. Vários loops podem normalmente ser reduzidos a um único loop. Por exemplo, o código a seguir:

```
for (int a = 0; a < MAX; ++a)
{
   // faça algo
}
// faça torradas com manteiga
for (int a = 0; a < MAX; ++a)
{
   // faça outra tarefa
}
```

provavelmente pode ser reduzido a:

```
for (int a = 0; a < MAX; ++a)
{
   // faça algo
   // faça outra tarefa
}
// faça torradas com manteiga
```

se fazer torradas com manteiga não depender de nenhum dos loops. Esse código não só é mais fácil de ler e de entender como também é provável que apresente um melhor desempenho, pois somente um loop deverá ser executado. Considere também as condicionais duplicadas e redundantes:

```
if (foo)
{
   if (foo && some_other_reason)
   {
```

```
        // a segunda verificação de foo é redundante
    }
}
```

Provavelmente, você não escreveria esse código de propósito, mas, depois de um pouco de trabalho de manutenção, muitos códigos acabam com uma estrutura descuidada como essa.

> **PONTO-CHAVE** Se você identificar uma duplicação, remova-a.

Recentemente, eu estava tentando depurar um device driver estruturado com dois loops principais de processamento. Ao inspecioná-los, vi que esses loops eram quase totalmente idênticos, com pequenas diferenças em relação ao tipo dos dados que estavam sendo processados. Esse fato não era imediatamente óbvio porque cada loop tinha 300 linhas (de código C bem denso)! Era um código tortuoso e difícil de acompanhar. Cada loop havia passado por um conjunto diferente de correções de bugs e, consequentemente, o código era instável e imprevisível. Um pouco de esforço para fatorar os dois loops em uma única versão reduziu imediatamente o espaço do problema pela metade; pude então me concentrar em um só local para encontrar e corrigir as falhas.

Código morto

Se você não fizer a manutenção de seu código, ele poderá definhar. E poderá morrer também. *Código morto* é um código que jamais será executado ou acessado. Não tem vida. Diga ao seu código que comece a viver ou desapareça.

Ambos os exemplos a seguir contêm seções de código morto que não são óbvias de imediato se você somente passar os olhos superficialmente pelo código:

```
if (size == 0)
{
    // ... vinte linhas de besteira ...
    for (int n = 0; n < size; ++n)
    {
        // este código jamais será executado
    }
```

```
    // ... mais vinte linhas de embromação ...
}
```

e

```
void loop(char *str)
{
    size_t length = strlen(str);
    if (length == 0) return;
    for (size_t n = 0; n < length; n++)
    {
        if (str[n] == '\0')
        {
            // este código jamais será executado
        }
    }
    if (length) return;
    // nem este
}
```

Outras manifestações de código morto incluem:

- funções que nunca são chamadas;
- variáveis que são escritas, mas não são lidas;
- parâmetros passados para um método interno que não são usados;
- enums, structs, classes ou interfaces que não são usados.

Comentários

Infelizmente, o mundo está repleto de comentários terríveis de código. Não podemos abrir um editor sem que tropecemos em alguns deles. O fato de muitos padrões corporativos de codificação serem uma pilha de lixo, obrigando a inclusão de milhões de comentários mortos, também não ajuda.

Um bom código *não* precisa de pilhas de comentários em que se sustentar ou para explicar o seu funcionamento. Uma escolha cuidadosa de nomes de variáveis, funções e classes e uma boa estrutura devem deixar o seu código totalmente claro. Duplicar todas essas informações em um conjunto de comentários representa uma redundância desnecessária. E como qualquer outra forma de duplicação, é também perigosa; é muito

fácil mudar um sem que o outro seja alterado.

Comentários estúpidos e redundantes variam do exemplo clássico de desperdício de bytes

```
++i; // incrementa i
```

a exemplos mais sutis, em que um algoritmo é descrito imediatamente antes do código

```
// percorre todos os itens em um loop e soma-os
int total = 0;
for (int n = 0; n < MAX; n++)
{
   total += items[n];
}
```

Bem poucos algoritmos, quando expressos no código, são complexos o suficiente para justificar esse nível de exposição. (Porém alguns *são* – aprenda a diferença!) Se um algoritmo precisar de comentários, eles serão mais bem fornecidos se fatorarmos a lógica em uma função nova e bem nomeada.

> **PONTO-CHAVE** Certifique-se de que todo comentário acrescente valor ao código. O código por si só diz *o que* e *como*. Um comentário deve explicar o *porquê* – mas somente se já não estiver claro.

Também é comum entrar em uma base de código antiga e ver um código "velho" que tenha sido cirurgicamente removido por meio de comentários. Não faça isso; é sinal de que alguém não teve coragem suficiente para realizar a extração cirúrgica completa ou que realmente não entendia o que estava sendo feito e achou que poderia ser necessário enxertar o código novamente depois. Remova o código completamente. É sempre possível tê-lo de volta depois a partir do sistema de controle de versões.

> **PONTO-CHAVE** Não remova códigos por meio de comentários. Isso confunde o leitor e atrapalha.

Não escreva comentários que descrevam para que o código *era* usado; não importa mais. Não coloque comentários no final de blocos de código ou de escopos; a estrutura do código deixa isso claro. E não crie uma arte

ASCII desnecessária.

Verbosidade

Muitos códigos são desnecessariamente prolixos. Na extremidade mais simples do espectro da verbosidade (que varia do infrarredundante ao ultravolúvel) está um código como este:

```
bool is_valid(const char *str)
{
   if (str)
      return strcmp(str, "VALID") == 0;
   else
      return false;
}
```

É bem extenso e, portanto, relativamente difícil perceber o seu propósito. Esse código pode ser facilmente reescrito como:

```
bool is_valid(const char *str)
{
   return str && strcmp(str, "VALID") == 0;
}
```

Não tenha medo do operador ternário se sua linguagem disponibilizar um; ele realmente ajuda a reduzir a desordem no código. Substitua este tipo de monstruosidade:

```
public String getPath(URL url) {
   if (url == null) {
      return null;
   }
   else {
      return url.getPath();
   }
}
```

por:

```
public String getPath(URL url) {
   return url == null ? null : url.getPath();
}
```

Declarações em estilo C (em que todas as variáveis são declaradas no início de um bloco e são usadas muito, muito depois) hoje em dia estão

ultrapassadas (a menos que você continue sendo forçado a usar oficialmente uma tecnologia defunta de compilador). O mundo evoluiu e o mesmo deve ocorrer com o seu código. Evite escrever algo como:

```
int a;
// ... vinte linhas de código C ...
a = foo();
// qual era mesmo o tipo de "a"?
```

Deixe as declarações de variáveis e as definições juntas para reduzir o esforço necessário para entender o código e diminuir os erros em potencial por causa de variáveis não inicializadas. Com efeito, às vezes, essas variáveis não fazem sentido de qualquer modo. Por exemplo:

```
bool a;
int b;
a = fn1();
b = fn2();
if (a)
   foo(10, b);
else
   foo(5, b);
```

pode facilmente se transformar no código menos extenso a seguir (e pode-se argumentar que seja mais claro):

```
foo(fn1() ? 10 : 5, fn2());
```

Design ruim

É claro que um código desnecessário não é somente o produto de erros de código no baixo nível ou de uma manutenção ruim. Pode ser causado por falhas de design no mais alto nível.

Um design ruim pode introduzir muitos caminhos desnecessários de comunicação entre os componentes – muito código extra para marshalling[1] de dados por nenhum motivo aparente. Quanto mais dados fluírem, mais provável será que eles sejam corrompidos no caminho.

Ao longo do tempo, os componentes de código se tornam redundantes ou podem mudar do uso original para algo bem diferente, deixando enormes seções de código não utilizado. Quando isso ocorrer, não tenha medo de

limpar todo o entulho. Substitua o componente antigo por um mais simples, que faça tudo o que for necessário.

Seu design deve considerar se já existem bibliotecas prontas que resolvam os seus problemas de programação. O uso dessas bibliotecas eliminará a necessidade de escrever muito código desnecessário. Como bônus, as bibliotecas populares provavelmente são robustas, extensíveis e bastante utilizadas.

Espaços em branco

Não entre em pânico! Não vou atacar os espaços em branco (ou seja, espaços, tabulações e quebras de linha). Espaços em branco são bons – não tenha medo de usá-los. Como uma pausa bem colocada ao recitar um poema, o uso sensato de espaços em branco ajuda a enquadrar o seu código.

O uso de espaços em branco normalmente não resulta em erros nem é desnecessário. Contudo você pode usar algo bom de forma exagerada, e vinte quebras de linha entre funções provavelmente será demais.

Considere também o uso de parênteses para agrupar construções lógicas. Às vezes, os parênteses ajudam a deixar a lógica mais clara, mesmo que eles não sejam necessários para se sobreporem à precedência dos operadores. Às vezes, eles são desnecessários e atrapalham.

Então o que devemos fazer?

Para ser justo, construções como essas em códigos antigos normalmente não são intencionais. Poucas pessoas se propõem a escrever deliberadamente um código trabalhoso, duplicado e sem sentido. (Mas há alguns programadores preguiçosos que continuamente optam pelo caminho mais curto em vez de investir tempo extra para escrever um bom código.) Com muita frequência, acabamos com esses problemas como legado de códigos que foram mantidos, estendidos, trabalhados e depurados por muitas pessoas durante um longo período de tempo.

Então o que podemos fazer a respeito? Devemos assumir a

responsabilidade. Não escreva códigos desnecessários e, quando trabalhar com um código "legado", preste atenção nos sinais de aviso. É hora de se tornar um militante. Reivindique nossos espaços em branco. Reduza a desordem. Faça uma limpeza. Restaure o equilíbrio.

Os porcos vivem em sua própria sujeira. Os programadores não precisam fazer isso. Faça sua própria limpeza. À medida que trabalhar em uma porção de código, remova todo o código desnecessário que você encontrar.

Esse é um exemplo de como seguir o conselho de Robert Martin e honrar a "regra dos escoteiros" no mundo da codificação: sempre deixe o acampamento mais limpo do que você o encontrou.[2]

> **PONTO-CHAVE** Todos os dias, deixe o seu código um pouco melhor do que estava. Elimine a redundância e a duplicação à medida que encontrá-las.

Contudo preste atenção nesta regra simples: faça as mudanças de "organização" separadas de outras mudanças funcionais. Isso garante que o que aconteceu ficará claro em seu sistema de controle de versões. Mudanças estruturais gratuitas misturadas com modificações funcionais são difíceis de seguir. E se houver um bug, será mais difícil descobrir se ele ocorreu por causa de sua nova funcionalidade ou da melhoria estrutural.

Conclusão

As funcionalidades de software não se correlacionam com o número de linhas de código nem com a quantidade de componentes de um sistema. Mais linhas de código não necessariamente significam mais software.

Portanto, se você não precisar de um código, não o escreva. Escreve menos código e ache algo mais divertido para fazer no lugar.

Perguntas

1. Você escreve naturalmente expressões lógicas sucintas? Suas expressões sucintas são tão concisas a ponto de serem incompreensíveis?
2. O *operador ternário* da família da linguagem C (por exemplo, `condition`

`? true_value : false_value`) torna as expressões mais ou menos legíveis? Por quê?

3. Devemos evitar *copiar e colar* código. Que nível de diferença deve haver em uma seção de código para que seja justificável não fatorá-la em uma função comum?
4. Como é possível identificar e remover códigos mortos?
5. Alguns padrões de codificação obrigam que toda função seja documentada com comentários de código especialmente formatados. Isso é útil? Ou é uma carga desnecessária, que introduz vários comentários extras sem valor?

Veja também

- *Melhore o código removendo-o* (Capítulo 4) – Descreve as técnicas para identificar seções maiores de código redundante e código morto e como removê-los.

> **TENTE ISTO...** Nos próximos dias, dê uma olhada em seu código de forma crítica e identifique qualquer seção redundante, duplicada ou longa demais. Trabalhe visando a remover esses códigos desnecessários.

---

1 N.T.: É o processo de transformação da representação de memória de um objeto em um formato de dados compatível para armazenamento ou transmissão; é usado normalmente quando os dados devem ser transferidos entre diferentes partes de um aplicativo ou de um aplicativo para outro (fonte: *https://pt.wikipedia.org/wiki/Marshalling*).

2 Robert C. Martin, *Código limpo: habilidades práticas do Agile software* (Alta Books, 2009).

CAPÍTULO 4

Melhore o código removendo-o

Atribuímos beleza àquilo que é simples, que não tenha partes supérfluas, que atenda exatamente ao seu propósito...

– Ralph Waldo Emerson

Menos é mais. É uma máxima banal, porém, às vezes, é realmente verdadeira.

Algumas das melhorias mais empolgantes de que me lembro de ter feito em um código envolveram a *remoção* de grandes porções dele. Se você me permite dizer, é uma sensação maravilhosa.

História de guerra: não há necessidade do código

Como uma equipe de desenvolvimento de software Agile, seguimos os princípios da sagrada eXtreme Programming, que inclui o *YAGNI*. Isso quer dizer *You Aren't Gonna Need It* (Você não irá precisar disso): um cuidado para não escrever código desnecessário – mesmo um código de que você *ache* que precisará em versões futuras. Não o escreva agora se não for precisar dele nesse momento. Espere até que surja uma necessidade genuína.

Isso soa como um conselho eminentemente sensato. E todos nós nos comprometemos com ele.

Porém, em virtude de a natureza humana ser como é, deixamos a desejar em algumas ocasiões. Certa vez, percebi que o produto estava demorando muito para executar determinadas tarefas – tarefas simples que deveriam ser quase instantâneas. Isso ocorria por causa de um excesso na implementação; ela continha detalhes extras que não eram necessários e estava cheia de hooks espalhados para futuras extensões. Nada disso estava sendo usado, porém, na época, cada um desses detalhes parecia ser uma adição razoável.

Sendo assim, simplifiquei o código, melhorei o desempenho do produto e reduzi o nível de entropia global do código simplesmente removendo todas as "funcionalidades" ofensoras do código. Felizmente, meus testes de unidade informaram que nada havia deixado de funcionar durante essa operação. Uma experiência simples e totalmente satisfatória.

PONTO-CHAVE Você pode aperfeiçoar um sistema adicionando um novo código. Você também pode melhorá-lo removendo códigos.

Indulgência no código

Por que todo esse código desnecessário acaba sendo escrito? Por que um programador sentiu a necessidade de escrever códigos extras, e como isso passou pela revisão de código ou pelos processos de verificação pelos pares?

É quase certo que foi a indulgência dos programadores com seus próprios vícios pessoais. Algo como:

- Era um código extra divertido e o programador quis escrevê-lo. *(Dica: escreva código porque ele acrescenta valor, e não porque você se diverte com ele ou porque gostaria de tentar escrevê-lo.)*
- Alguém achou que era uma funcionalidade necessária no futuro, portanto decidiu codificá-la agora enquanto pensava nela. *(Dica: isso não está de acordo com o YAGNI. Se não precisar do código agora, não o escreva.)*
- Mas era só um código pequeno, e não uma funcionalidade "extra" enorme. Era mais fácil simplesmente implementá-lo agora, em vez de retornar ao cliente para ver se a funcionalidade realmente era necessária. *(Dica: escrever e manter um código extra sempre exige mais tempo. E o cliente é bem acessível. Uma pequena quantidade de código vira uma bola de neve com o passar do tempo e se transforma em um trabalho enorme, que exige manutenção.)*
- O programador inventou requisitos extras que não estavam documentados para justificar a funcionalidade extra. O requisito, na verdade, era falso. *(Dica: os programadores não definem os requisitos do*

sistema; os clientes é que o fazem.)

Tínhamos um processo de desenvolvimento enxuto e bem consolidado, desenvolvedores muito bons e verificações procedimentais definidas para evitar esses tipos de situação. E, apesar disso, códigos extras desnecessários continuavam surgindo sorrateiramente.

É surpreendente, não é mesmo?

Não é ruim, é inevitável

Mesmo que você possa evitar a adição de *novas* funcionalidades desnecessárias, porções de código morto continuarão a surgir naturalmente durante o desenvolvimento de seu software. Não se sinta envergonhado com isso! Esses códigos surgem devido a diversas origens acidentais e inevitáveis, incluindo:

- Funcionalidades foram removidas da interface de usuário da aplicação, porém o código de suporte no backend foi mantido. Esse código jamais será chamado novamente. Necrose instantânea. Geralmente ele não é removido "porque podemos precisar dele no futuro e deixá-lo ali não fará mal a ninguém".
- Tipos de dados ou classes que não estão mais sendo usados tendem a permanecer no projeto. Não é fácil dizer que você está removendo a última referência a uma classe quando estiver trabalhando em uma parte diferente do projeto. *Partes* de uma classe também podem ser consideradas obsoletas: por exemplo, você pode retrabalhar os métodos de modo que uma variável-membro não seja mais necessária.
- Funcionalidades legadas em produtos *raramente* são removidas. Mesmo que seus usuários não as queiram mais e jamais as utilizem novamente, remover funcionalidades de um produto não passa uma boa impressão. Isso reduziria a lista incrível de funcionalidades assinaladas como prontas. Desse modo, incorremos em um overhead perpétuo para testar funcionalidades do produto que jamais serão usadas novamente.
- A manutenção do código durante seu período de vida faz com que

seções de uma função não sejam executadas. Pode ser que um loop jamais execute uma iteração porque o código adicionado imediatamente antes negou uma invariante ou que blocos de código condicionais jamais sejam executados. Quanto mais antiga a base de código, mais disso você verá. O C disponibiliza prestativamente o pré-processador como um ótimo sistema para escrever códigos macarrônicos não executáveis.

- Código de UI gerado por assistentes (wizards) inserem hooks que, com frequência, jamais são usados. Se um desenvolvedor acidentalmente der um clique duplo em um controle, o assistente adicionará o código de backend, porém o programador não chegará nem perto da implementação. Dá mais trabalho remover esses tipos de blocos de código gerados automaticamente do que simplesmente ignorá-los e fingir que eles não existem.
- Muitas funções retornam valores que não são usados nunca. Todos nós sabemos que é moralmente condenável ignorar o código de erro de uma função e *jamais* faríamos isso, certo? Porém muitas funções são escritas para *fazer algo* e retornam um resultado que alguém *poderá* achar útil. Ou não. Não é um código de erro; é somente uma pequena informação. Por que dispender esforço extra para calcular o valor de retorno e escrever testes para ele se ninguém jamais irá usá-lo?
- Muito código de “debug” é um código necrosado. Muitos códigos de suporte não serão necessários depois que a implementação inicial for concluída. É uma estrutura desagradável, que esconde a beleza da arquitetura subjacente. Não é incomum ver inúmeros printouts para diagnóstico e verificações de invariantes inativos, testes de pontos de hook e outros códigos semelhantes que jamais serão usados novamente. Eles congestionam o código e tornam a manutenção mais difícil.

E daí?

Isso realmente importa? Certamente, devemos aceitar que um código morto seja inevitável e não devemos nos preocupar muito se o projeto

continuar funcionando. Qual é o custo de um código desnecessário?

- Não podemos negar que um código desnecessário, assim como qualquer outro código, exige manutenção ao longo do tempo. Custa tempo e dinheiro.
- Um código extra também dificulta conhecer o projeto e exige mais compreensão e navegação adicional.
- Classes com um milhão de métodos que podem ou não ser usados são impenetráveis e somente incentivam um uso desleixado, em vez incentivar uma programação cuidadosa.
- Mesmo que você adquira o computador mais rápido que o dinheiro possa comprar e tenha o melhor conjunto de ferramentas de compilação, um código morto irá deixar seus builds mais lentos, fazendo com que você seja menos produtivo.
- É mais difícil refatorar, simplificar ou otimizar o seu programa quando ele estiver infestado de códigos zumbis.

Um código morto não o matará, mas tornará sua vida mais difícil do que o necessário.

> **PONTO-CHAVE** Remova códigos mortos sempre que for possível. Eles atrapalham e tornam o seu desenvolvimento lento.

Despertando os mortos

Como você pode encontrar códigos mortos?

A melhor abordagem é prestar atenção quando você estiver trabalhando com o código. Seja responsável pelas suas ações e não se esqueça de sempre fazer uma limpeza quando terminar o seu trabalho. Revisões regulares de código ajudam a identificar códigos mortos.

Se você tiver uma atitude séria em relação a eliminar seções de código não utilizadas, há algumas ferramentas ótimas que percorrem o código e que mostrarão exatamente os locais em que os problemas se encontram.[1] Bons IDEs, especialmente quando usados com linguagens de tipos estáticos, podem identificar códigos não usados automaticamente. Para APIs

públicas, muitos IDEs têm um recurso de “encontrar referências” que pode mostrar se uma função é chamada alguma vez.

Para identificar funcionalidades mortas, você pode instrumentalizar o seu produto e reunir métricas sobre o que os clientes realmente utilizam. Isso é útil para tomar todo tipo de decisão de negócios em vez de simplesmente identificar um código não utilizado.

Extração cirúrgica

Não há nenhum problema em remover um código morto. Faça uma amputação. Não significa que você estará jogando algo fora. Sempre que você perceber que uma funcionalidade antiga será novamente necessária, ela poderá ser facilmente acessada em seu sistema de controle de versões.

PONTO-CHAVE É seguro remover códigos dos quais você *possa* precisar no futuro. É sempre possível recuperá-los do sistema de controle de versões.

Entretanto há um contra-argumento para essa visão simples (e verdadeira): como os novos recrutas saberão que o código removido está disponível no sistema de controle de versões se eles não souberem que o código existia, antes de tudo? O que os impedirá de escrever sua própria versão (com bugs ou incompleta)? Essa é uma preocupação válida. Porém, de modo semelhante, o que os impediria de reescrever sua própria versão se eles simplesmente não perceberem que o fragmento de código já está em outro lugar?

Como nos capítulos anteriores, lembre-se de remover códigos mortos em um único passo; não misture isso com um check-in no sistema de controle de versões que também adicione outras funcionalidades. Sempre separe seu trabalho de “limpeza” de outras tarefas de desenvolvimento. Isso torna o histórico de versões mais claro e também faz com que ressuscitar um código removido seja muito fácil.

PONTO-CHAVE A limpeza do código sempre deve ser feita em commits separados das modificações funcionais.

Conclusão

Códigos mortos surgem até mesmo nas melhores bases de código. Quanto maior o projeto, mais código morto você terá. Não é um sinal de falha. Porém não fazer algo a respeito quando um código morto for encontrado *é* sinal de falha. Quando você descobrir que há códigos que não estão sendo usados ou descobrir um caminho no código que não poderá ser executado, remova esse código desnecessário.

Ao escrever uma nova porção de código, não modifique a especificação. Não acrescente funcionalidades "pequenas" que você ache que sejam interessantes, mas que ninguém tenha pedido. Será bem fácil acrescentá-las no futuro caso sejam necessárias. Mesmo que acrescentá-las *pareça* ser uma boa ideia. Não faça isso.

Perguntas

1. Como você pode identificar um "código morto", que não está sendo executado em seu programa?
2. Se você remover temporariamente um código que não seja necessário no momento (mas que poderá sê-lo no futuro), você deverá deixá-lo comentado (para que ele continue visível) no código-fonte ou deverá simplesmente apagá-lo por completo (pois ele ficará armazenado no histórico de revisões)? Por quê?
3. A remoção de funcionalidades legadas (não usadas) é sempre o certo a fazer? Há algum risco inerente à remoção de seções de código? Como você pode determinar o momento certo para remover funcionalidades não utilizadas?
4. Qual é o percentual da base de código de seu projeto atual que você acha que é desnecessário? Sua equipe tem uma cultura de acrescentar itens de que seus membros *gostem* ou que eles *achem* que será útil?

Veja também

- *Escreva menos código!* (Capítulo 3) – Discute a remoção de duplicações no nível micro, ou seja, a remoção de linhas de código desnecessárias.

- *Chafurdando na lama* (Capítulo 7) – Como percorrer um caminho em um código problemático para que você possa identificar o que deve ser removido.
- *Lidando com a complexidade* (Capítulo 12) – A redução de código morto diminui a complexidade de seu software.
- *Controle eficiente de versões* (Capítulo 20) – Remover um código morto não significa que esse código estará perdido para sempre. Você pode recuperá-lo do sistema de controle de versões caso um erro seja cometido.

> **TENTE ISTO...** Procure códigos mortos e desnecessários nos arquivos em que você estiver trabalhando. Remova-os!

1 Pode ser que você já as tenha – dê uma olhada nas opções de avisos (warnings) disponibilizadas pelo seu compilador.

CAPÍTULO 5

O fantasma de um código do passado

Viverei no Passado, no Presente e no Futuro.
Os Espíritos dos Três florescerão em mim.
Não deixarei de lado as lições que eles me ensinaram!

– Charles Dickens
Um conto de Natal

A nostalgia não é o que costumava ser. E nem o seu código antigo. Quem sabe quais espíritos funcionais e demônios tipográficos espreitam em seu antigo trabalho? Você achou que ele era perfeito quando o criou – porém lance um olhar crítico sobre o seu antigo código e você inevitavelmente trará à luz todo tipo de truque com o código.

Os programadores, como uma espécie, se esforçam para progredir. Amamos aprender técnicas novas e empolgantes, encarar novos desafios e solucionar problemas mais interessantes. É natural. Considerando a alta rotatividade no mercado de trabalho e a duração média dos contratos na área de programação, não é de surpreender que poucos desenvolvedores de software permaneçam com a mesma base de código por um longo período de tempo.

Mas o que isso provoca no código que geramos? Que tipo de atitude isso promove em nosso trabalho? Eu defendo que programadores excepcionais são definidos mais pela sua atitude em relação ao código que escrevem e pela maneira como o escrevem do que pelo código propriamente dito.

O programador médio tende a não manter *seu próprio* código por muito tempo. Em vez de chafurdar na própria lama, vamos em busca de novas pastagens e rolamos na lama de *outra pessoa*. Bonito. Temos até mesmo a tendência de deixar nossos "projetos de estimação" de lado à medida que nossos interesses evoluem.

É claro que é divertido reclamar do código pobre de outras pessoas, porém facilmente nos esquecemos de como o nosso próprio trabalho era ruim. E você jamais escreveria um código ruim *intencionalmente*, certo?

Rever o seu código antigo pode ser uma experiência esclarecedora. É como visitar um parente distante que está envelhecendo e que você não vê com muita frequência. Logo você descobrirá que não o conhece tão bem quanto imaginava. Você havia se esquecido de fatos sobre ele, de suas manias engraçadas e de seus modos irritantes. E ficará surpreso em ver como essa pessoa mudou desde a última vez em que você a viu (talvez para pior).

PONTO-CHAVE Rever o seu código antigo mostrará as melhorias (ou não) em suas habilidades de codificação.

Ao rever um código antigo que você tenha criado, você poderá sentir um estremecimento por diversos motivos.

Apresentação

Muitas linguagens permitem efetuar uma interpretação artística no layout de indentação do código. Apesar de algumas linguagens terem um padrão consagrado pelo uso para o estilo de apresentação, continua havendo uma gama enorme de questões relacionadas ao layout que você se verá explorando ao longo do tempo. Qual deles permanecerá depende das convenções de seu projeto atual ou de sua experiência após anos de experimentos.

Tribos diferentes de programadores C++, por exemplo, seguem esquemas diferentes de apresentação. Alguns desenvolvedores adotam o esquema da biblioteca-padrão:

```
struct standard_style_cpp
{
   int variable_name;
   bool method_name();
};
```

Alguns têm uma experiência mais ao estilo Java:

```
struct JavaStyleCpp
```

```
{
   int variableName;
   bool methodName();
};
```

E outros seguem um modelo C#:

```
struct CSharpStyleCpp
{
   int variableName;
   bool MethodName();
};
```

É uma diferença simples, porém que afeta profundamente o seu código de diversas maneiras.

Outro exemplo em C++ relaciona-se ao layout de listas de membros que efetuem inicializações. Uma de minhas equipes passou deste esquema tradicional:

```
Foo::Foo(int param)
: member_one(1),
  member_two(param),
  member_three(42)
{
}
```

para um estilo que insere vírgulas como separadores no início da linha seguinte, desta maneira:

```
Foo::Foo(int param)
: member_one(1)
, member_two(param)
, member_three(42)
{
}
```

Identificamos uma série de vantagens com o último estilo (é mais fácil "eliminar" partes centrais com o uso de macros de pré-processador ou de comentários, por exemplo). Esse esquema de vírgulas como prefixo pode ser empregado em diversas situações de layout (por exemplo, em vários tipos de lista: membros, enumerações, classes base e outros), proporcionando um formato consistente e interessante. Também há desvantagens, e um dos principais problemas citados é que ele não é tão "comum" quanto o estilo anterior de layout. Os layouts automáticos

default dos IDEs também tendem a brigar com esse estilo.

Sei que ao longo dos anos meu próprio estilo de apresentação mudou bastante conforme a empresa em que eu estivesse trabalhando na época.

Desde que um estilo seja empregado de forma consistente em sua base de código, essa é uma preocupação realmente trivial e não há nada com que você deva se sentir envergonhado. Estilos individuais de codificação raramente fazem muita diferença depois que você se acostuma com eles, porém estilos inconsistentes de codificação em um projeto podem deixar mais lento o trabalho de qualquer pessoa.

O estado da arte

A maioria das linguagens rapidamente desenvolveu suas próprias bibliotecas. Com o passar dos anos, as bibliotecas Java cresceram, passando de algumas centenas de classes úteis para uma verdadeira miríade de classes, com variantes diferentes da biblioteca, de acordo com o alvo em que o Java fosse implantado. Nas versões do C#, sua biblioteca-padrão também floresceu. À medida que as linguagens evoluem, suas bibliotecas agregam mais recursos.

E à medida que essas bibliotecas evoluem, algumas das partes mais antigas se tornam obsoletas.

Uma evolução como essa (que é especialmente rápida no início da vida de uma linguagem) infelizmente pode deixar o seu código anacrônico. Qualquer pessoa que estiver lendo o seu código pela primeira vez poderá presumir que você não entendia a nova linguagem nem os recursos da biblioteca, mas esses recursos simplesmente não existiam quando o código havia sido escrito.

Por exemplo, quando o C# adicionou os genéricos, o código que você escreveria desta maneira:

```
ArrayList list = new ArrayList(); // sem tipo
list.Add("Foo");
list.Add(3); // opa!
```

com seu potencial inerente para bugs, se transformou em:

```
List<string> list = new List<string>();
list.Add("Foo");
list.Add(3); // erro de compilação - bom
```

Há um exemplo bem semelhante em Java, com nomes de classe surpreendentemente semelhantes!

O estado da arte se transforma de modo muito mais rápido que o seu código. Especialmente se for um código antigo, sem manutenção.

Mesmo a biblioteca C++ (relativamente conservadora) evoluiu consideravelmente a cada nova versão. Os recursos da linguagem C++11 e a biblioteca de suporte fez com que muito código C++ antigo parecesse fora de moda. A introdução de um modelo de threading suportado pela linguagem fez com que as bibliotecas de thread de terceiros (normalmente implementadas com APIs bastante questionáveis) se tornassem redundantes. A introdução de lambdas eliminou a necessidade de vários códigos "trampolim" extensos e manualmente implementados. O `for` baseado em intervalos ajuda a eliminar várias árvores sintáticas para que você possa ver o design do código com mais clareza. Depois que você começa a usar esses recursos, retornar ao código mais antigo sem eles dará a impressão de termos dado um passo para trás.

Idioms

Cada linguagem, com seu conjunto único de construções e os recursos de biblioteca, tem um método de uso em particular que é considerado a "melhor prática". São os *idioms* adotados pelos usuários experientes, ou seja, os modos de uso que foram aperfeiçoados e que conquistaram a preferência ao longo do tempo.

Esses idioms são importantes. São o que os programadores experientes esperam ler; são formas familiares que permitem focar no design geral do código em vez de se atolar em preocupações de código de nível macro. Normalmente, os idioms formalizam padrões que evitam erros comuns ou bugs.

Talvez o mais embaraçoso seja rever códigos antigos e ver o quanto eles não eram idiomáticos. Se agora você conhece melhor os idioms aceitos na

linguagem com a qual você está trabalhando, seu velho código não idiomático poderá parecer muito, muito errado.

Há alguns anos, trabalhei com uma equipe de programadores C que estava passando (bem, fazendo lentamente algumas misturas) para o admirável mundo novo (na época) do C++. Uma de suas primeiras adições à nova base de código foi uma macro auxiliar `max`:

```
#define max(a,b) ((a)>(b)) ? (a) : (b))
// você sabe dizer por que temos todas esses parênteses?

void example()
{
   int a = 3, b = 10;
   int c = max(a, b);
}
```

Em algum momento, alguém revisitou esse código inicial e, conhecendo melhor o C++, percebeu como era ruim. O código foi reescrito em C++ mais idiomático, como mostrado a seguir, o que corrigiu alguns bugs muito sutis à espreita:

```
template <typename T>
inline T max(const T &a, const T &b)
{
   // Veja mamãe! Não há necessidade de parênteses!
   return a > b ? a : b;
}

void better_example()
{
   int a = 3, b = 10;

   // isto não funcionaria com a macro
   // porque ++a seria avaliado duas vezes
   int c = max(++a, b);
}
```

A versão original também apresentava outro problema: a reinvenção da roda. A melhor solução é simplesmente usar a função `std::max` pronta, que sempre existiu. Depois que a vemos, ela se torna óbvia:

```
// não declare nenhuma função max

void even_better_example()
{
   int a = 3, b = 10;
```

```
    int c = std::max(a,b);
  }
```

É o tipo de situação que o deixaria arrepiado agora, se você retornasse a esse código. Mas você não tinha a mínima ideia do idiom correto naquela época.

Esse é um exemplo simples, porém, à medida que as linguagens obtêm novos recursos (por exemplo, lambdas), o tipo de código idiomático que você escreveria hoje poderia ser bem diferente das gerações de código anteriores.

Decisões de design

Eu *realmente* escrevi aquilo em Perl? O que eu tinha na cabeça?! Eu *realmente* usei um algoritmo de ordenação tão simplista? Eu *realmente* escrevi todo aquele código manualmente, em vez de usar uma função pronta da biblioteca? Eu *realmente* criei um acoplamento entre aquelas classes de modo tão desnecessário? Eu *realmente* não poderia ter criado uma API mais limpa? Eu *realmente* deixei o gerenciamento de recursos a cargo do código do cliente? Posso ver muitos bugs e falhas em potencial espreitando por aí!

À medida que aprender mais, você perceberá que há maneiras melhores de formular o seu design no código. É a voz da experiência quem está dizendo. Você comete alguns erros, lê alguns códigos diferentes, trabalha com programadores talentosos e logo percebe que melhorou suas habilidades de design.

Bugs

Talvez esse seja o motivo pelo qual você seja arrastado para uma base de código antiga. Às vezes, voltar com um olhar renovado revela problemas óbvios que você não havia percebido na época. Depois de ter sido mordido por determinados tipos de bug (geralmente aqueles dos quais você é afastado ao usar idioms comuns), você começará naturalmente a ver bugs em potencial em códigos antigos. É o *sexto sentido* do

programador.

Conclusão

Nenhum remorso compensa a oportunidade de uma vida desperdiçada.

– Charles Dickens

Um conto de Natal

Revisitar seu código antigo é como fazer uma revisão de código para si mesmo. É um exercício valioso; talvez você deva fazer um passeio rápido por partes de seus antigos trabalhos. Você gosta da maneira como costumava programar? Quanto você aprendeu desde então?

Esse tipo de percepção realmente *importa*? Se o seu código antigo não for perfeito, mas funcionar, você deve fazer algo a respeito? Você deve voltar lá e "ajustar" o código? Provavelmente não – *se não estiver com problemas, não corrija*. O código não apodrece, a menos que o mundo em volta dele mude. Seus bits e bytes não se degradam, portanto o significado provavelmente permanecerá constante. Ocasionalmente, uma atualização no compilador ou na linguagem ou em uma biblioteca de terceiros poderá "quebrar" o seu código antigo, ou, quem sabe, uma mudança de código em outro lugar possa invalidar uma pressuposição feita por você. Contudo, normalmente, o código resistirá bravamente, mesmo que não seja perfeito.

É importante apreciar como os tempos mudaram, como o mundo da programação evoluiu e como suas habilidades pessoais melhoraram com o passar do tempo. Deparar-se com um código antigo que não pareça "certo" é bom: mostra que você aprendeu e se aperfeiçoou. Talvez você não tenha a oportunidade de revisá-lo agora, porém saber de onde você veio ajuda a determinar para onde você está indo em sua carreira de codificação.

Assim como o Espírito do Natal Passado, há lições interessantes a serem aprendidas sobre precaução com o seu código antigo se você passar um tempo observando-o.

Perguntas

1. Como o seu código antigo se enquadra no mundo moderno? Se ele não parecer tão ruim, isso significa que você não aprendeu nada de novo recentemente?
2. Há quanto tempo você está trabalhando com sua linguagem principal? Quantas versões do padrão da linguagem ou de bibliotecas prontas foram introduzidas nesse período? Que recursos da linguagem foram introduzidos e que moldaram o estilo do código que você escreve?
3. Considere alguns dos idioms comuns que você emprega naturalmente hoje em dia. Como eles ajudam a evitar erros?

Veja também

- *Mantendo as aparências* (Capítulo 2) – Contém mais discussões sobre layout de código.
- *Nada está gravado a ferro e fogo* (Capítulo 18) – O código nunca permanece imutável, nem o seu entendimento dele.
- *Um conto de dois sistemas* (Capítulo 13) – Contém um exemplo em que um código antigo é revisitado; mostra como aprender com os erros e apreciar os sucessos.

TENTE ISTO... Faça um tour rápido por alguns de seus trabalhos antigos. Você gosta da maneira como costumava programar? Quanto você aprendeu desde então?

10.000 MACACOS
(OU ALGO POR AÍ)

FANTASMAS DO PASSADO

OTTO VON BISMARCK

O BLÁ DIÁRIO

1 DE ABRIL DE 1976

DESASTRE COM MARTELOS

Todos os participantes descalços da convenção Faça Você Mesmo foram admitidos em um hospital após um acidente em massa grotesco relacionado a martelos e pés.
Um porta-voz da convenção disse: "Francamente,

"SOMENTE UM TOLO APRENDE COM SEUS PRÓPRIOS ERROS."

"O HOMEM SÁBIO APRENDE COM OS ERROS DOS OUTROS."

10.000 MACACOS
(OU ALGO POR AÍ)

CAPÍTULO 6

Percorrendo um caminho

... a Investigação de Coisas difíceis pelo Método da Análise deve sempre preceder o Método da Composição.

– Sir Isaac Newton

Um novo recruta juntou-se à minha equipe de desenvolvimento. Nosso projeto, embora não fosse vasto, era relativamente grande e continha várias áreas diferentes. Havia muito para aprender antes que ele pudesse se tornar eficiente. Como ele poderia traçar um caminho pelo código? Como iniciante, de que modo ele poderia se tornar produtivo rapidamente?

É uma situação comum; uma com a qual nos deparamos de tempos em tempos. Se você não passou por isso, deverá ver mais códigos e mudar para projetos novos com mais frequência. (É importante não ficar estagnado, trabalhando somente com uma base de código e uma equipe o tempo todo.)

Abordar qualquer base vasta de código já existente é difícil. Você deve rapidamente:

- descobrir por onde deve começar a olhar o código;
- descobrir o que cada seção do código faz, e como o faz;
- avaliar a qualidade do código;
- descobrir como navegar pelo sistema;
- entender os idioms do código para que suas alterações se encaixem naturalmente;
- descobrir a provável localização de qualquer funcionalidade (e os bugs consequentes causados por ela);
- entender o relacionamento do código com suas partes satélites

importantes (por exemplo, seus testes e a documentação).

Você deve aprender isso rapidamente, pois não vai querer que suas primeiras alterações sejam embaraçosas demais nem duplicar um trabalho existente de forma acidental ou fazer com que outras partes deixem de funcionar.

Uma pequena ajuda de meus amigos

Meu novo colega tinha uma vantagem maravilhosa nesse processo de aprendizado. Ele se juntou a um grupo em que as pessoas já conheciam o código, que podiam responder a inumeráveis perguntas pequenas sobre ele e apontar o local em que as funcionalidades existentes poderiam ser encontradas. Esse tipo de ajuda simplesmente tem um valor inestimável.

Se você puder trabalhar junto com alguém que já tenha experiência com o código, explore isso. Não tenha medo de fazer perguntas. Se puder, aproveite as oportunidades para programar junto com alguém e/ou ter suas alterações revisadas.

> **PONTO-CHAVE** Seu melhor caminho pelo código é ser conduzido por alguém que já conheça o terreno. Não tenha medo de pedir ajuda!

Se não puder importunar as pessoas ao redor, não tenha medo; pode haver pessoas prestativas um pouco além. Procure fóruns online ou listas de emails que tenham informações úteis e pessoas prestativas. Geralmente, há uma comunidade saudável que cresce em torno de projetos populares de código aberto.

O truque ao pedir ajuda é sempre ser educado e grato. Faça perguntas sensatas e apropriadas. “Você pode fazer a minha lição de casa?” jamais terá uma boa resposta. Esteja sempre preparado para ajudar os outros com informações em troca.

Use o bom senso: não se esqueça de procurar uma resposta para a sua pergunta no Google antes. É simplesmente uma questão de boa educação não fazer perguntas tolas que poderiam ser facilmente pesquisadas por conta própria. As pessoas não gostarão de você se você ficar fazendo perguntas básicas o tempo todo e desperdiçar o tempo precioso delas.

Como o menino que gritava que o lobo estava lá e que não conseguiu ajuda quando realmente precisou, uma série de perguntas idiotas fará com que seja menos provável receber ajuda em casos mais complexos quando for necessário.

Procure pistas

Se você estiver se atolando nas profundezas sombrias de um sistema de software sem ter uma orientação pessoal, procure pistas que o guiarão pelo código.

A seguir temos alguns bons indicadores:

Facilidade de obter o código-fonte

Qual é o nível de dificuldade de obter o código-fonte?

É somente um checkout único e simples do sistema de controle de versões, cujos arquivos poderão ser colocados em qualquer diretório em seu computador de desenvolvimento? Ou você deverá fazer checkout de várias partes separadas e instalá-las em locais específicos de seu computador?

Paths de arquivo fixos no código são perversos. Eles impedem que você crie facilmente versões diferentes do código.

> **PONTO-CHAVE** Projetos saudáveis exigem um único checkout para a obtenção de toda a base de código, e o código poderá ser colocado em qualquer *diretório* em seu computador de build. Não dependa de vários passos de checkout ou de código com localizações fixas.

Além da disponibilidade do código-fonte, considere também a disponibilidade das *informações sobre* a saúde do código. Há um servidor de build para CI (Continuous Integration, ou Integração contínua) que garanta continuamente que todas as partes do código sejam geradas com sucesso? Há resultados publicados para qualquer teste automatizado?

Facilidade de criar um build do código

Isso pode ser muito informativo. Se for difícil gerar um build do código, geralmente será difícil trabalhar com ele.

O build depende de ferramentas incomuns que deverão ser instaladas? (Até que ponto essas ferramentas estão atualizadas?)

Qual é o nível de dificuldade de gerar um build do código do zero? Há uma documentação simples e adequada no próprio código? O build do código é feito diretamente do sistema de controle de versões ou você deve realizar manualmente vários ajustes pequenos na configuração antes do build?

Um único passo simples cria o build de todo o sistema ou isso exige vários passos individuais de build? O processo de build exige intervenção manual?[1] Você pode trabalhar com uma pequena parte do código e fazer o build somente dessa seção ou deve refazer o build de todo o projeto repetidamente para trabalhar com um componente pequeno?

> **PONTO-CHAVE** Um build saudável é executado em um só passo, sem intervenção manual durante o processo.

Como um build de disponibilização de versão é feito? É o mesmo processo dos builds de desenvolvimento ou você deve seguir um conjunto bem diferente de passos?

Quando o build é executado, ele é discreto? Ou há muitos e muitos avisos que podem encobrir problemas mais insidiosos?

Testes

Procure os testes: testes de unidade, de integração, fim a fim (end-to-end) e outros semelhantes. Há algum? Que parte da base do código está em teste? Os testes são executados automaticamente ou exigem um passo adicional do build? Com que frequência os testes são executados? Que nível de cobertura eles proporcionam? Eles parecem ser apropriados e bem definidos ou há somente alguns stubs simples para parecer que há uma boa cobertura de testes no código?

Há uma relação quase universal aqui: um código com uma boa suíte de testes normalmente também é um código bem fatorado, bem planejado e tem um bom design. Esses testes atuam como um ótimo caminho pelo código em teste, ajudando você a entender a interface do código e os padrões de uso. Também é um bom lugar para começar a

trabalhar na correção de um bug (você pode começar adicionando um teste de unidade simples em falha – então corrija esse teste sem deixar que os outros parem de funcionar).

Estrutura de arquivos

Observe a estrutura de diretórios. Ela corresponde ao formato do código? Revela claramente as áreas, os subsistemas ou as camadas do código? É organizada? As bibliotecas de terceiros estão separadas de forma organizada do código do projeto ou tudo é um emaranhado confuso?

Documentação

Procure a documentação do projeto. Há alguma? Ela está bem redigida? Está atualizada? Talvez a documentação esteja escrita no próprio código com *NDoc*, *Javadoc*, *Doxygen* ou um sistema parecido. O quão abrangente e atualizada parece estar essa documentação?

Análises estáticas

Execute ferramentas no código para determinar a saúde e identificar as associações. Há algumas ferramentas boas disponíveis para navegação em códigos-fonte, e o Doxygen também pode gerar diagramas de classe e de fluxos de controle muito úteis.

Requisitos

Há algum documento original de requisitos do projeto ou de especificações funcionais? (De acordo com minha experiência, esses documentos normalmente tendem a ostentar pouca relação com o produto final, mas, apesar disso, são documentos históricos interessantes.) Há uma wiki do projeto em que os conceitos comuns estão reunidos?

Dependências do projeto

O código utiliza frameworks específicos e bibliotecas de terceiros? Que nível de informações você deve ter sobre eles? Você não pode conhecer todos os seus aspectos desde o início, em especial porque algumas bibliotecas são enormes (Boost, estou olhando para você). No entanto vale a pena ter uma noção dos recursos oferecidos e onde procurá-los.

O código faz bom uso da biblioteca-padrão da linguagem? Ou muitas rodas foram reinventadas? Desconfie de códigos com seu próprio conjunto de classes personalizadas para coleções ou primitivas de thread domésticas. Um código disponibilizado pelo núcleo do sistema provavelmente será mais robusto, mais bem testado e livre de bugs.

Qualidade do código

Navegue pelo código para ter uma ideia de sua qualidade. Observe a quantidade e a qualidade dos comentários no código. Há muitos códigos mortos – códigos redundantes comentados, deixados para apodrecer? O estilo de codificação é consistente por toda parte?

É difícil formar uma opinião conclusiva a partir de uma investigação breve como essa, porém você poderá ter rapidamente uma ideia razoável de como está uma base de código a partir de uma leitura básica.

Arquitetura

A essa altura, você deverá ser capaz de ter uma noção razoável do formato e da modularização de seu sistema. Você consegue identificar as principais camadas? As camadas estão claramente separadas ou elas estão entrelaçadas? Há uma camada de banco de dados? O quão razoável ela parece ser? Você consegue ver o esquema? Ele é organizado? Como a aplicação se comunica com o mundo externo? Qual é a tecnologia de GUI? E de I/O de arquivos? E de rede?

O ideal é que a arquitetura de um sistema seja um conceito de mais alto nível que você conheça antes de fazer explorações mais detalhadas. Entretanto isso não é o que ocorre normalmente, e você *descobrirá* a verdadeira arquitetura à medida que se aprofundar no código.

> **PONTO-CHAVE** Com frequência, a *verdadeira* arquitetura de um sistema difere do design *ideal*. Sempre confie no código, e não na documentação.

Promova uma *arqueologia do software* em qualquer código que pareça questionável. Explore os logs do sistema de controle de versões e execute "svn blame" (ou algum comando equivalente) para ver a origem e a

evolução de algumas das confusões. Procure ter uma ideia da quantidade de pessoas que trabalharam no código no passado. Quantas delas continuam na equipe?

Aprenda fazendo

Uma mulher precisa de um homem tanto quanto um peixe precisa de uma bicicleta.

– Irina Dunn

Você pode ler quantos livros quiser sobre a teoria de andar de bicicleta. Pode estudar as bicicletas, desmontá-las, montá-las novamente e investigar a física e a engenharia por trás delas. Mas seria o mesmo que aprender a andar com um peixe. Até subir em uma bicicleta, colocar os pés nos pedais e tentar andar de verdade, você não avançará. Você aprenderá mais caindo algumas vezes do que lendo durante dias sobre como se equilibrar.

O mesmo ocorre com códigos.

Ler um código irá levá-lo somente até certo ponto. Você só conhecerá realmente uma base de código se subir e tentar andar com ela, cometendo erros e caindo. Não deixe que a falta de atividade impeça você de prosseguir. Não erga uma barreira intelectual que o impeça de trabalhar com o código.

Já vi muitos bons programadores inicialmente paralisados devido à falta de confiança em seu próprio entendimento.

Deixe isso de lado. Mergulhe de cabeça. Seja ousado. Modifique o código.

> **PONTO-CHAVE** A melhor maneira de conhecer um código é modificando-o. Aprenda então com os seus erros.

O que você deve modificar?

À medida que estiver conhecendo o código, procure locais em que você poderá imediatamente gerar um benefício, mas que as chances de que algo deixe de funcionar sejam mínimas (ou de escrever um código embaraçoso).

Procure qualquer parte que o fará percorrer o sistema.

Frutos ao alcance das mãos

Tente tarefas simples e pequenas como identificar um bug menor que tenha uma correlação direta com um evento para iniciar a sua procura (por exemplo, uma atividade na GUI). Comece com um relatório de falha pequeno, repetível e de baixo risco em vez de escolher um pesadelo substancial e intermitente.

Inspecione o código

Passe a base de código por alguns validadores (por exemplo, *Lint*, *Fortify*, *Cppcheck*, *FxCop*, *ReSharper* ou algo semelhante). Verifique se os avisos do compilador foram desabilitados; habilite-os novamente e corrija as mensagens. Isso fará você conhecer a estrutura do código e dará uma pista sobre a sua qualidade.

Corrigir esse tipo de aviso geralmente não é complicado e vale muito a pena: é um ótimo ponto de partida. Com frequência, isso fará você percorrer boa parte do código rapidamente. Esse tipo de mudança não funcional no código mostra como tudo se encaixa e a localização de cada parte. Você terá uma boa noção do zelo dos desenvolvedores, e as partes do código que demandem mais preocupação e que exijam cuidados extras serão enfatizadas.

Estude e depois aja

Estude uma pequena porção do código. Faça críticas. Determine se há pontos fracos. Refatore-a. Sem piedade. Dê nomes adequados às variáveis. Transforme seções vastas de código em funções menores e adequadamente nomeadas.

Alguns desses exercícios lhe darão uma boa noção da maleabilidade do código e da facilidade de efetuar correções e modificações. (Já vi bases de código que realmente se rebelaram contra refatorações).

Tome cuidado: escrever código é mais fácil que lê-lo. Muitos programadores, em vez de se esforçarem para ler e entender um código existente, preferem dizer que ele "está feio" e reescrevê-lo. Isso certamente os ajuda a ter um entendimento mais profundo do código, porém à custa

de muita mudança de código desnecessária, tempo perdido e, muito provavelmente, novos bugs.

Teste antes

Dê uma olhada nos testes. Descubra como acrescentar um novo teste de unidade e como adicionar um novo arquivo de teste em uma suíte. Como os testes são executados?

Um ótimo truque é tentar adicionar um único teste de uma linha em falha. A suíte de teste falha imediatamente? Esse *teste de fumaça* (smoke test) prova que os testes não estão sendo ativamente ignorados.

Os testes servem para mostrar como cada componente funciona? Eles mostram bem os pontos de interface?

Organize a casa

Dê uma lustrada na interface de usuário. Faça melhorias simples na UI que não afetem a funcionalidade principal, mas que tornem a aplicação mais agradável de ser usada.

Organize os arquivos-fontes: corrija a hierarquia de diretórios. Faça com que ela corresponda à organização do IDE ou dos arquivos de projeto.

Documente o que você encontrar

O código tem algum tipo de arquivo de documentação *README* de mais alto nível que explique como começar a trabalhar com ele? Se não tiver, crie um e inclua o que você aprendeu até agora nesse arquivo.

Peça a um dos programadores mais experientes para revisá-lo. Isso mostrará até que ponto o seu conhecimento está correto e também ajudará outros novatos.

À medida que você compreender melhor o sistema, mantenha um diagrama de camadas das principais seções do código. Mantenha-o atualizado enquanto estiver aprendendo. Você descobriu que o sistema está bem dividido em camadas, com interfaces claras entre elas e que não há acoplamentos desnecessários? Ou percebeu que as seções do código estão interconectadas sem necessidade? Procure maneiras de introduzir

interfaces para promover uma separação sem alterar as funcionalidades existentes.

Se não houver nenhuma descrição da arquitetura até então, seu diagrama poderá servir como uma documentação para conduzir um novo recruta pelo sistema.

Conclusão

Investigações científicas são como guerras travadas na saleta ou no sofá contra todos os contemporâneos e precursores de alguém.

– Thomas Young

Quanto mais você se exercitar, menos dor você sentirá e maiores serão os benefícios adquiridos. Com a codificação não é diferente. Quanto mais você trabalhar em novas bases de código, maior será a sua capacidade de entender um código novo de forma eficiente.

Perguntas

1. Você acessa novas bases de código com frequência? Você acha fácil percorrer um código que não lhe seja familiar? Quais são as ferramentas comuns que você usa para investigar um projeto? Quais ferramentas poderão ser acrescentadas a esse arsenal?
2. Descreva algumas estratégias para adicionar um código novo em um sistema que você ainda não compreenda totalmente. Como você pode colocar um firewall em torno do código existente para protegê-lo (e a você)?
3. O que pode ser feito para que seja mais fácil a um novo recruta entender o código? O que você deve fazer para melhorar o estado de seu projeto atual?
4. O tempo provável que você permanecerá trabalhando no código no futuro afeta o seu esforço e a maneira pela qual você conhece um código existente? É mais provável que você faça uma correção "rápida e suja" em um código no qual você não precisará mais dar manutenção, mesmo que outras pessoas tenham de fazê-lo depois? Isso é apropriado?

Veja também

- *Chafurdando na lama* (Capítulo 7) – Como avaliar a qualidade do código e fazer ajustes seguros.
- *Viva para amar o aprendizado* (Capítulo 24) – Conhecer uma nova base de código é como conhecer qualquer assunto novo. Essas técnicas irão ajudar você.
- *Nada está gravado a ferro e fogo* (Capítulo 18) – Aprenda fazendo; faça alterações no código para entendê-lo melhor.

> **TENTE ISTO...** Na próxima vez que você abordar um código novo, planeje cuidadosamente um caminho por ele. Utilize essas técnicas para ter um bom entendimento.

---

1 Um passo único e automatizado de build significa que esse poderá ser colocado em uma estrutura de CI e executado automaticamente.

CAPÍTULO 7

Chafurdando na lama

Assim como os cães retornam ao seu próprio vômito, os tolos repetem suas imprudências.

– Provérbios 26:11

Todos nós já nos deparamos com ele: um *código areia movediça*. Você entra nele sem saber e logo tem aquela sensação de que está afundando. O código é denso, não é maleável e resiste a qualquer esforço para movê-lo. Quanto mais esforços você investe, mais para o fundo você é engolido. É a armadilha para humanos da era digital.

Como um programador eficiente aborda um código que, para dizermos de forma polida, *não é tão bom assim*? Quais são as nossas estratégias para *lidar com uma porcaria*?

Não entre em pânico, vista suas calças à prova de areia e, bem, entre nela...

Fareje os sinais

Alguns códigos são ótimos, como uma obra de arte sofisticada ou uma poesia bem composta. Têm uma estrutura discernível, cadências reconhecíveis, uma métrica ritmada, além de coerência e beleza, características que os tornam agradáveis de ler e que fazem ser prazeroso trabalhar com eles.

Infelizmente, porém, esse nem sempre é o caso.

Alguns códigos são confusos e desestruturados: um zigue-zague de `goto`s que encobrem qualquer semelhança com um algoritmo. Alguns são difíceis de ler: têm um layout pobre e uma nomenclatura descuidada. Alguns códigos são amaldiçoados com uma estrutura rígida desnecessária: acoplamento terrível e pouca coesão. Alguns códigos têm

fatoração ruim: entrelaçam códigos de UI com uma lógica de baixo nível. Alguns códigos estão repletos de duplicações: fazem com que o projeto seja maior e mais complexo do que o necessário, além de abrigar o mesmo bug várias vezes. Alguns códigos cometem "abuso de OO": usam herança pelos motivos mais inapropriados e associam profundamente partes do código que não teriam nenhuma necessidade de estar acoplados. Alguns códigos são como uma ave perniciosa no ninho: usam C# escrito em estilo JavaScript.

Alguns códigos têm ainda outras perversidades insidiosas: um comportamento instável, em que uma mudança em um local faz com que um módulo aparentemente não relacionado falhe – a condição exata da *teoria do caos no código*. Alguns códigos sofrem de comportamento pobre em threading: empregam primitivas inadequadas de thread ou mostram uma total falta de entendimento do uso seguro e concorrente dos recursos. Esse problema pode ser bem difícil de ser identificado, reproduzido e diagnosticado, pois se manifesta de modo intermitente.

(Sei que eu não deveria reclamar, mas, às vezes, juro que os programadores deveriam ser proibidos de digitar a palavra *thread* sem antes obter uma licença para empunhar uma arma tão perigosa quanto essa.)

> **PONTO-CHAVE** Esteja preparado para se deparar com códigos ruins. Encha sua caixa de ferramentas com armas afiadas para lidar com eles.

Para trabalhar de modo eficiente com códigos desconhecidos, você deve ser capaz de identificar rapidamente esses tipos de problema e saber como responder.

Entrando na fossa

O primeiro passo consiste em fazer uma avaliação realista da cena do crime no código. Você chegou às margens de um novo código. Em que você *está* entrando?

O código pode ter sido fornecido a você com um estigma anterior. Ninguém quer mexer nele porque sabem que ele é traiçoeiro. É um código

que você descobre ser uma areia movediça ao se ver afundando nele.

É muito fácil dispensar um código novo porque ele não está escrito no estilo que você gostaria que estivesse. É realmente um trabalho ruim? É um *código areia movediça* de verdade ou você simplesmente não está familiarizado com ele? Não faça julgamentos precipitados sobre o código ou sobre os autores que o criaram até ter investido algum tempo investigando.

Tome cuidado para não tornar isso pessoal.

Saiba que poucas pessoas se propõem a escrever um código de baixa qualidade. Alguns códigos ruins foram simplesmente escritos por um programador menos capacitado. Ou por um programador hábil em um dia ruim. Depois de ter aprendido uma nova técnica ou de ter conhecido o idiom preferido de uma equipe, um código que parecia ser perfeitamente adequado um mês atrás será um emaranhado confuso agora e exigirá refatoração.

Não espere que nenhum código, nem mesmo o seu, seja perfeito.

> **PONTO-CHAVE** Acabe com a sensação de repulsa ao se deparar com um código "ruim". Em vez disso, procure maneiras práticas de melhorá-lo.

A avaliação diz que...

Já vimos as técnicas para navegar em uma base de código nova no capítulo 6.

À medida que construir um modelo mental de uma nova porção do código, comece a avaliar a sua qualidade usando benchmarks como:

- As APIs externas são claras e razoáveis?
- Os tipos usados foram bem escolhidos e têm nomes adequados?
- O layout do código é organizado e consistente? (Embora isso, certamente, não seja uma garantia da qualidade do código subjacente, acho que um código inconsistente e confuso também tende a estar mal estruturado e é difícil trabalhar com ele. Programadores que buscam um código de alta qualidade e maleável também tendem a se importar

com uma apresentação limpa e clara. Contudo não baseie o seu julgamento somente na apresentação.)

- A estrutura dos objetos em cooperação é simples e pode ser vista claramente? Ou o controle flui de forma imprevisível pela base de código?
- Você consegue determinar facilmente a localização do código que gera um determinado efeito?

Pode ser difícil realizar essa avaliação inicial. Talvez você não conheça a tecnologia envolvida ou o domínio do problema. Você pode não estar familiarizado com o estilo de codificação.

Considere o emprego da *arqueologia de software* em sua avaliação: vasculhe os logs do sistema de controle de versões à procura de pistas sobre a qualidade. Determine a idade desse código. O quanto ele é mais velho em relação a todo o projeto? Quantas pessoas trabalharam nele ao longo do tempo? Quando ele foi modificado pela última vez? Há algum colaborador que tenha trabalhado recentemente no projeto? Você pode pedir informações a ele a respeito do código? Quantos bugs foram encontrados e corrigidos nessa área? Muitas correções de bugs nesse local indicam que o código é pobre.

Trabalhando em terreno arenoso

Você identificou um código areia movediça e agora está em estado de alerta. Você precisa de uma estratégia sólida para trabalhar com ele.

Qual é o plano de ataque apropriado?

- Você deve corrigir o código ruim?
- Deve fazer o mínimo de ajustes necessários para resolver o seu problema atual e depois se afastar desse código?
- Deve remover o código necrosado e substituí-lo por um trabalho novo e melhor?

Olhe para a sua bola de cristal. Geralmente, a resposta correta será dada pelos seus planos futuros. Quanto tempo você trabalhará com essa seção

de código? Saber que você irá armar acampamento e trabalhar nesse local por um tempo influenciará o nível de investimento a ser feito. Não tente uma reescrita completa se você não tiver o tempo necessário para isso.

Além disso, considere a frequência com que esse código tem sido modificado até agora. Os analistas financeiros dirão que *o desempenho passado não é um indicador de resultados futuros*. Mas geralmente é. Invista o seu tempo com sabedoria. Esse código pode ser desagradável, porém, se ele funcionou de forma adequada durante anos sem ajustes, provavelmente não será apropriado "organizá-lo" agora, em especial se for pouco provável que você vá fazer muitas outras alterações no futuro.

> **PONTO-CHAVE** Escolha suas batalhas. Considere cuidadosamente se você deve investir tempo e esforço "organizando" um código ruim. Pode ser prático deixá-lo como está no momento.

Se você concluir que não é apropriado embarcar em um retrabalho massivo do código nesse momento, isso não quer dizer que você será necessariamente deixado à deriva em um mar de esgoto. Você pode lutar para reconquistar parte do controle sobre o código fazendo limpezas progressivamente.

Limpando a sujeira

Independentemente de você estar fazendo alterações visando ao longo prazo ou de estar efetuando apenas uma pequena correção, preste atenção no conselho de Robert Martin e siga a "regra dos escoteiros": *sempre deixe o acampamento mais limpo do que você o encontrou*. Pode ser que não seja apropriado efetuar uma melhoria completa hoje, porém isso não quer dizer que você não possa fazer do mundo um local um pouco menos terrível.

> **PONTO-CHAVE** Siga a regra dos escoteiros. Sempre que mexer em um código, deixe-o *melhor* do que estava.

Essa mudança pode ser simples: cuidar de um layout inconsistente, corrigir um nome de variável inadequado, simplificar uma cláusula condicional complexa ou dividir um método longo em subfunções

menores e mais bem nomeadas.

Se você revisitar uma seção do código regularmente e, a cada vez, deixá-la um pouco melhor do que estava, não irá demorar muito para ter algo que poderá ser classificado como *bom*.

Fazendo ajustes

O conselho único mais importante ao trabalhar com um código confuso é este:

> **PONTO-CHAVE** Faça alterações no código lentamente e com cuidado. Faça uma alteração de cada vez.

Isso é tão importante que eu gostaria que você parasse, retornasse e lesse esse conselho novamente.

Há várias maneiras práticas de seguir esse conselho. De modo específico:

- Não mude o layout do código enquanto estiver ajustando funcionalidades. Organize o layout se for necessário. Em seguida, faça um commit de seu código. Somente *então* faça as mudanças funcionais. (No entanto é preferível preservar o layout existente, a menos que ele esteja tão ruim a ponto de estar atrapalhando.)
- Faça tudo o que puder para garantir que sua "organização" preserve o comportamento existente. Utilize ferramentas automatizadas de confiança ou (se elas não estiverem disponíveis), revise e inspecione cuidadosamente as suas alterações; solicite pares extras de olhos para realizar essa tarefa. Esta é a principal diretriz para a *refatoração*: ter um conjunto bem conhecido de técnicas para melhorar a estrutura do código.

  Esse objetivo pode ser alcançado de modo eficiente somente se o código estiver coberto por um conjunto sólido de testes de unidade. É provável que um código confuso não tenha nenhum teste criado, portanto considere se você deve inicialmente escrever alguns testes para capturar comportamentos importantes do código.
- Ajuste as APIs que encapsulam o código sem modificar diretamente a

lógica interna. Corrija os nomes, os tipos de parâmetro e a ordenação; introduza consistência de modo geral. Insira, quem sabe, uma nova interface externa – a interface que você gostaria que o código tivesse. Implemente-a de acordo com a API existente. No futuro, você poderá retrabalhar o código por trás dessa interface.

Tenha coragem de mudar o código. Você tem uma rede de proteção: o sistema de controle de versões. Se você cometer um erro, sempre será possível voltar no tempo e tentar novamente. Provavelmente, não será um desperdício de esforço, pois você conhecerá o código e a sua capacidade de adaptação ao fazer isso.

Às vezes, vale a pena eliminar o código e substituí-lo de forma ousada. Um código com manutenção ruim, que não tenha sido organizado nem refatorado, pode ser muito difícil de ser corrigido por partes. Há, porém, um perigo inerente em substituir um código inteiro; a confusão ilegível de casos especiais *pode* estar presente por algum motivo. Cada remendo e hack de código contém uma parte importante da funcionalidade, revelada por meio de uma experiência amarga. Ignore esses comportamentos sutis por sua conta e risco.

Um livro excelente que lida com mudanças apropriadas em códigos areia movediça é *Trabalho eficaz com código legado* de Micheal Feathers.[1] Nesse livro, o autor descreve técnicas sólidas para a introdução de códigos que se encaixem em códigos existentes – locais em que você pode incluir pontos de teste de forma saudável e segura.

História de guerra: o curioso caso do código de contêiner

Era uma vez uma classe contêiner. Ela era central em nosso projeto. Internamente, era péssima. A API também era horrível. O programador original havia trabalhado arduamente para acabar com os erros do código. Os bugs contidos estavam ocultos pelo comportamento já confuso. Na verdade, o comportamento confuso *já era*, por si só, um bug.

Um de nossos programadores – um desenvolvedor altamente qualificado – tentou refatorar e corrigir esse contêiner. Ele deixou a interface externa intacta e melhorou vários aspectos internos: garantiu a correção dos métodos, revisou o comportamento indevido do tempo de vida dos objetos, verificou o

desempenho e procurou dar mais elegância ao código.

Um código horrível, feio, simplista e estúpido foi removido e substituído pelo oposto. Porém, em seu esforço de preservar a API antiga, essa nova versão era muito artificial internamente, parecendo-se mais com um projeto de ciências do que com um código útil. Era difícil trabalhar com ele. Embora o código expressasse o comportamento antigo (bizarro) de forma sucinta, não havia espaço para extensões.

Tivemos dificuldade em trabalhar com essa nova versão também. Havia sido um esforço em vão.

Posteriormente, outro desenvolvedor simplificou a maneira como usávamos o contêiner: os requisitos mais estranhos foram eliminados, o que simplificou a API. Foi um ajuste relativamente simples no projeto. No contêiner, removemos alguns trechos de código. A classe se tornou mais simples, menor e mais fácil de ser verificada.

Às vezes, você deve pensar de forma mais ampla para garantir as melhorias corretas.

Código ruim? Programadores ruins?

Sim, é frustrante ser freado por um código ruim. O programador eficiente não só lida bem com um código ruim, mas também com as pessoas que o escreveram. Sair atribuindo culpa pelos problemas no código não irá ajudar. As pessoas não tendem a escrever códigos ruins de propósito.

PONTO-CHAVE Não é preciso sair atribuindo culpa por um código “ruim”.

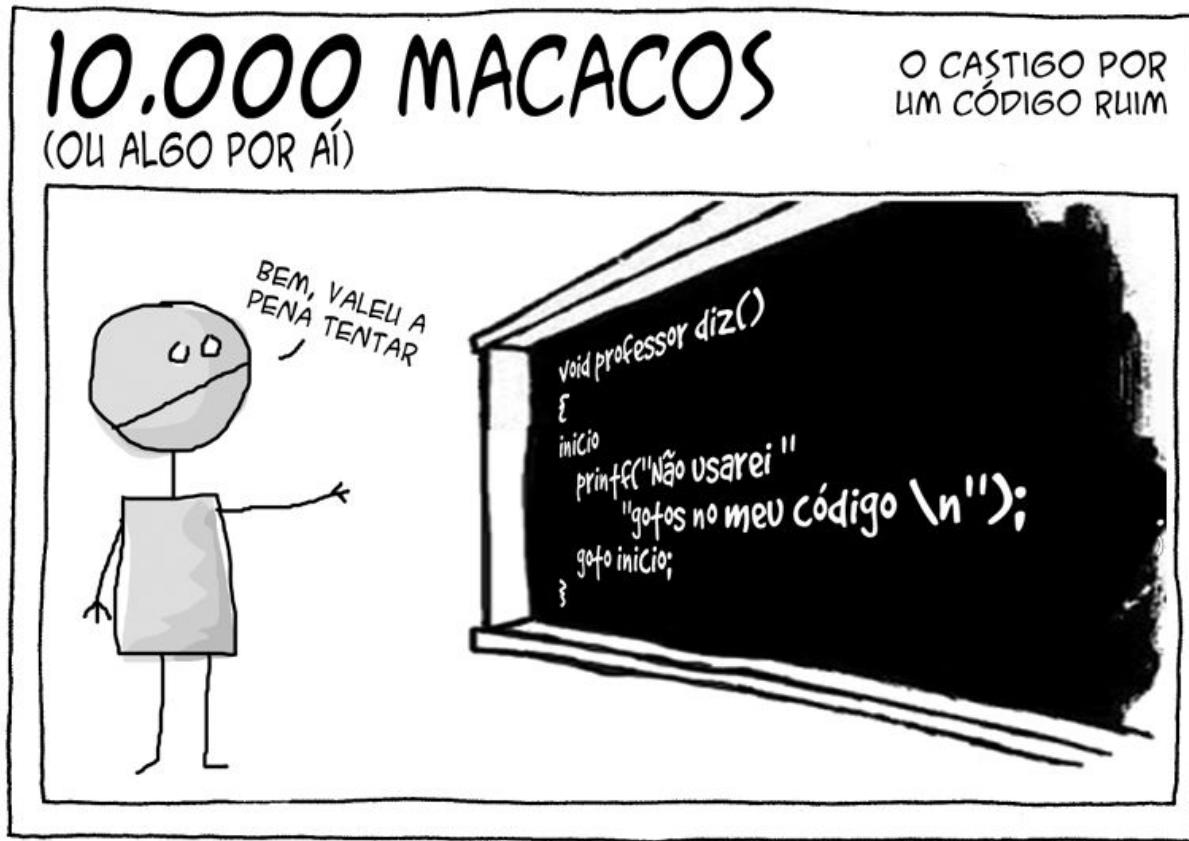
Talvez o autor original não entendesse a utilidade da refatoração de código ou não visse uma maneira de expressar a lógica de forma ordenada. É igualmente provável que haja outros pontos semelhantes que você ainda não compreenda. Quem sabe os programadores se sentissem pressionados a trabalhar depressa e tiveram de criar atalhos (acreditando na mentira de que isso os ajudaria a chegar lá mais rápido; raramente é o que acontece).

Mas é claro que você está acima disso.

Se puder, *aprecie* a chance de organizar o código. Pode ser muito recompensador bagunçar a estrutura e a sua sanidade. Em vez de ver isso como um exercício maçante, encare como uma chance de introduzir mais qualidade.

Trate isso como se fosse uma lição. Aprenda. Como você evitará a repetição desses mesmos erros de codificação?

Avalie a sua atitude à medida que fizer essas melhorias. Pode ser que você ache que saiba mais que o autor original. Mas você sempre sabe mais?

Já vi essa história se repetir várias vezes: um programador júnior "corrigiu" o trabalho de um programador mais experiente com a mensagem de commit "código refatorado para melhorar a organização". O código realmente parecia estar mais organizado. Porém funcionalidades importantes haviam sido removidas. O autor original posteriormente reverteu as mudanças com uma mensagem de commit: "código refatorado funcionando novamente".

Perguntas

1. Por que o código se torna confuso com tanta frequência?
2. Como podemos evitar que isso aconteça, antes de tudo? É possível?
3. Quais são as vantagens de fazer alterações no layout de modo separado das alterações no código?
4. Quantas vezes você já se deparou com um código desagradável? Com que frequência esse código *realmente* era ruim, e não somente um código que "não estava de acordo com o seu gosto"?

Veja também

- *Percorrendo um caminho* (Capítulo 6) – Técnicas para se familiarizar com uma nova base de código.
- *Melhore o código removendo-o* (Capítulo 4) – Melhore programas "sujos" exorcizando códigos mortos.
- *Uma ode ao código* (Capítulo 38) – Uma reação desnecessariamente extrema a um código ruim.

TENTE ISTO... Empregue a regra dos escoteiros. Melhore cada porção do código que você mexer, mesmo que seja somente por uma fração.

---

1 Michael Feathers, *Trabalho eficaz com código legado* (Bookman, 2013).

CAPÍTULO 8

Não ignore esse erro!

Tudo de que você precisa é de ignorância e de confiança e o sucesso estará garantido.

– Mark Twain

Prepare-se para uma história de dormir apócrifa. Uma parábola sobre um programador, se você me permite...

Certa noite, eu estava andando por uma rua para encontrar alguns amigos em um bar. Fazia tempo que não bebíamos uma cerveja juntos e eu estava ansioso para vê-los novamente. Em minha pressa, não prestei atenção no caminho. Tropecei no canto da calçada e caí de cara no chão. Bem, isso foi merecido por não estar atento, pensei eu.

Machuquei a minha perna, mas eu estava com pressa para encontrar meus amigos. Então me levantei e continuei. Quanto mais eu caminhava, mais a dor piorava. Embora eu a ignorasse inicialmente, pensando ser apenas o choque, logo percebi que havia algo errado.

Apesar disso, continuei apressado em direção ao bar. Eu estava em agonia no momento em que cheguei lá. Não tive uma noite muito agradável porque eu estava terrivelmente distraído. Na manhã seguinte, fui ao médico e descobri que havia fraturado a tíbia. Se eu tivesse parado quando senti a dor, teria evitado um bocado de danos a mais, provocados pela caminhada. Provavelmente, foi a pior manhã de minha vida...

Muitos programadores escrevem códigos como minha noite desastrosa.

Erro? Que erro? Não é sério. É verdade. Posso ignorá-lo. Essa *não* é uma estratégia vencedora para um código sólido. Com efeito, é apenas pura preguiça. (Do tipo ruim.) Não importa o quão improvável você ache que seja um erro em seu código, sempre verifique e trate-o. Sempre. Se não o fizer, você não estará economizando tempo; estará armazenando

problemas em potencial para o futuro.

> **PONTO-CHAVE** Não ignore possíveis erros em seu código. Não adie o tratamento de erros para "mais tarde" (você não poderá desviar-se dele).

Maneiras de informar erros

Informamos erros em nosso código de diversos modos, incluindo:

Códigos de retorno

Uma função retorna um valor. Alguns deles querem dizer "não funcionou". É muito fácil ignorar códigos de erro de retorno. Você não verá nada no código que enfatize o problema. Na verdade, ignorar o valor de retorno de algumas funções-padrão em C tornou-se uma prática comum. Com que frequência você verifica o valor de retorno de um `printf`?

Os códigos de retorno talvez sejam o canal de informação de erro mais popular que existe: vemos *funções* retornarem valores, *processos* do sistema operacional retornarem valores e, em alguns sistemas, até as *threads* podem retornar valores.

Normalmente, esse código é um valor inteiro; por convenção, zero quer dizer sucesso e um valor diferente de zero corresponde a um código de erro. Em códigos modernos, é um idiom bem estranho, e podemos escrever um código bem mais expressivo retornando uma *tupla* de valores ou um tipo "opcional" para indicar sucesso e o valor em um tipo (por exemplo, o tipo `boost::optional` em C++ ou `Nullable<T>` em C#). Linguagens de programação funcionais podem informar erros de acordo com o *tipo* de retorno da função em vez de usar um valor mágico; o Haskell disponibiliza a classe `Maybe`, o Scala oferece `Option` e `Either`.

Efeitos colaterais

`errno` – o garoto-propaganda dos efeitos colaterais – é uma aberração curiosa da linguagem C: uma variável global diferente, usada para indicar um erro. É fácil de ser ignorada, difícil de usar e resulta em todo tipo de problema desagradável – por exemplo, o que acontece se

você tiver várias threads chamando a mesma função?

Você poderá ver outros canais secundários ou efeitos colaterais usados para sinalizar erros. Por exemplo, pode haver outra função que deva ser chamada para verificar o "estado relativo ao sucesso" ou um objeto pode passar para um estado "inválido" quando algo der errado.

Exceções

As exceções constituem uma maneira mais estruturada e suportada pela linguagem para sinalizar e tratar erros. E você não poderá ignorá-las. Ou pode? Já vi muito código como este:

```
try
{
   // ...faz algo...
}
catch (...) {} // ignora erros
```

O que salva nessa construção horrível é que ela enfatiza o fato de que você está fazendo algo moralmente duvidoso.

As exceções não são perfeitas. Seus críticos reclamam que elas ocultam o caminho do erro. Uma exceção pode se desenrolar por um método e provocar repercussões terríveis (por exemplo, causar leak de recursos ou deixar de satisfazer o que uma função se propõe). Mas como esse método não contém nenhum código de tratamento de erro para si mesmo, você não perceberá esses problemas.

Assim como ocorre com várias outras tecnologias, o uso eficiente de exceções exige bastante disciplina. Isso está muito além do escopo deste capítulo.

PONTO-CHAVE Use as exceções de forma adequada e disciplinada. Entenda os idioms e os requisitos de sua linguagem para fazer um uso eficiente delas.

A loucura

Não tratar erros resulta em:

Código frágil

Esse tipo de código está cheio de falhas difíceis de identificar.

Código sem segurança

Os crackers geralmente exploram tratamentos pobres de erro para invadir sistemas de software.

Estrutura ruim

Se houver erros em seu código que sejam tediosos de lidar continuamente, é provável que você tenha uma interface ruim. Expresse-os de uma maneira melhor para que os erros não sejam tão onerosos.

Assim como você deve verificar todos os erros em potencial em seu código, exponha todas as condições de erro em potencial em suas interfaces. Não as oculte nem finja que seus serviços funcionarão sempre.

Os programadores devem ter ciência dos erros de programação. Os usuários devem ter ciência dos erros de uso.

Não basta somente fazer um *log* do erro (em algum lugar) e esperar que um operador diligente perceba-o e faça algo a seu respeito algum dia. Quem tem conhecimento da existência do log? Quem o verifica? Quem é a pessoa mais provável de fazer algo a respeito? Se encerrar o programa não for uma opção, garanta que os problemas sejam marcados de maneira discreta, porém óbvia, para não serem ignorados.

As desculpas

Por que não verificamos os erros? Há inúmeras desculpas comuns. Com quais destas você concorda? Como você considera cada uma delas?

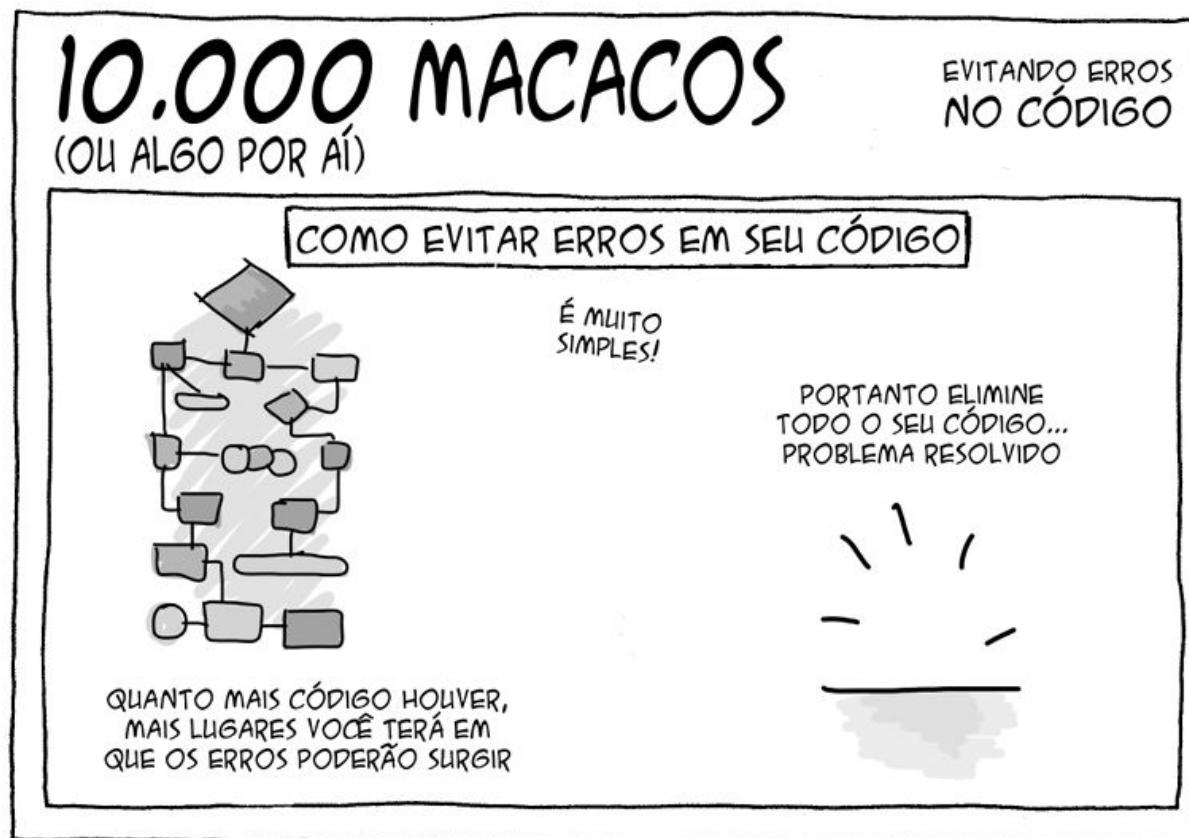
- O tratamento de erros congestiona o fluxo do código, fazendo com que seja difícil lê-lo e mais difícil ainda identificar o fluxo “normal” da execução.
- É um trabalho extra e eu tenho um prazo prestes a se esgotar.
- Sei que essa chamada de função *jamais* retornará um erro (`printf` sempre funciona, `malloc` sempre retorna mais memória e, se esses falharem, nós teremos problemas mais sérios, de qualquer maneira...).
- É somente um programa para brincar e não precisa ser implementado

em um nível semelhante ao de produção.

- Minha linguagem não me incentiva a fazê-lo. (Por exemplo, a filosofia do Erlang é "deixe falhar": códigos incorretos devem falhar *rapidamente* e fazer com que o processo Erlang seja encerrado. Bons sistemas Erlang são projetados para serem robustos contra processos que falhem, portanto o tratamento de erros não é tão importante assim.)

Conclusão

Este é um capítulo bem curto. Poderia ser muito, muito mais extenso. Mas fazer isso seria um erro. A mensagem é simples: Não ignore os erros.

Perguntas

1. Como você pode garantir que seu código não irá ignorar os erros informados pelos níveis mais baixos? Considere as soluções de nível de código e as técnicas relacionadas a processos.
2. As exceções não podem ser tão facilmente ignoradas quanto os códigos de retorno. Isso as torna um sistema mais seguro para informar erros?
3. Quais abordagens são necessárias para trabalhar com um código que misture códigos de erro e exceções?
4. Quais técnicas de teste ajudarão a identificar um código que falhe devido a um tratamento inadequado de erros?

Veja também

- *Amor pelas linguagens* (Capítulo 29) – O estilo apropriado para informar e tratar erros depende da linguagem que estiver em uso.
- *Espere pelo inesperado* (Capítulo 9) – As condições de erro são um exemplo do tipo de situações "inesperadas" que devemos considerar para que o nosso código seja robusto.

TENTE ISTO... Programe uma revisão do código com que se trabalha com mais frequência em seu sistema. Determine quantas condições de erro foram deixadas sem tratamento. Em seguida, revise alguns códigos sem

manutenção frequente e compare os resultados.

CAPÍTULO 9

Espere pelo inesperado

Sempre Alerta... o sentido do lema é que um escoteiro deve estar preparado ao pensar com antecedência e praticar o modo de agir em qualquer acidente ou emergência para que jamais seja pego de surpresa.

– Robert Baden-Powell

Dizem que algumas pessoas veem o copo meio cheio enquanto outras o veem meio vazio. No entanto a maioria dos programadores não vê nenhum copo; eles escrevem códigos que simplesmente não consideram situações incomuns. Eles não são nem otimistas nem pessimistas. Não são nem mesmo realistas. São *ignoralistas*.

Ao escrever um código, não considere somente o fluxo de execução que você espera que vá ocorrer. A cada passo, considere todas as ocorrências *incomuns* que possam surgir, independentemente do quão *pouco provável* você ache que elas sejam.

Erros

Qualquer função que você chamar poderá não funcionar conforme esperado.

- Se estiver com sorte, a função retornará um código de erro para indicar isso. Em caso afirmativo, você deve verificar esse valor; jamais o ignore.
- A função poderá lançar uma exceção se ela não puder cumprir o que promete. Garanta que o seu código irá lidar com uma exceção que se propague por ele. Independentemente de você capturar a exceção e tratá-la ou deixar que ela passe adiante na pilha de chamadas, certifique-se de que seu código esteja correto. Isso inclui não causar

leak de recursos nem deixar o programa em um estado inválido.

- Ou a função poderá não retornar nenhuma indicação de falha, porém não fará o que você espera nem irá informá-lo. Você pede a uma função que exiba uma mensagem. Ela sempre irá exibi-la? Às vezes, ela poderá falhar e consumir a mensagem?

Sempre considere erros dos quais você pode se recuperar e escreva um código apropriado de recuperação. Considere também os erros dos quais você não possa se recuperar. Escreva o seu código para fazer o melhor possível – não ignore simplesmente o erro.

Certifique-se de que o seu tratamento de erros seja idiomático e use os métodos adequados da linguagem. O Erlang, por exemplo, tem uma filosofia do tipo "deixe falhar", em que um código defensivo não é incentivado em favor de um código que deixe os erros causarem falhas evidentes e visíveis a serem tratados no nível de um processo.

Threading

O mundo evoluiu, partindo de aplicações com uma única thread para ambientes mais complexos, geralmente com várias threads. Interações incomuns entre partes do código são comuns nesse caso. É difícil enumerar todos os possíveis entrelaçamentos entre os caminhos tomados pelo código, quanto mais reproduzir uma interação problemática em particular mais de uma vez.

Para reduzir esse tipo de imprevisibilidade, certifique-se de que você compreenda os princípios básicos da concorrência e como desacoplar threads para que elas não interajam de formas perigosas. Conheça os sistemas rápidos e confiáveis de transmissão de mensagens entre contextos de threads sem que condições de concorrência sejam introduzidas ou que as threads sejam desnecessariamente bloqueadas.

Encerramento

Planejamos a construção de um sistema: como criar todos os objetos, como fazer todos os pratos girarem e como manter esses objetos

executando e esses pratos girando. Menos atenção é dedicada à outra extremidade do ciclo de vida: como interromper o código de modo elegante sem causar leak de recursos, travar ou provocar falhas.

Encerrar o seu sistema e destruir todos os objetos é especialmente difícil em um sistema com várias threads. Quando sua aplicação for encerrada e os objetos de trabalho forem destruídos, garanta que um objeto não ficará tentando usar outro que já tenha sido destruído. Não enfileire callbacks de threads que tenham objetos já descartados por outras threads como alvo.

A moral da história

O “inesperado” não é o incomum. É o material com que os bugs são feitos. Você deve escrever o seu código com isso em vista.

É importante pensar nesses problemas bem cedo no desenvolvimento de seu código. Você não pode atingir esse estado de correção como um pensamento *a posteriori*; os problemas são insidiosos e se alojam profundamente em seu código. Demônios como esses são difíceis de exorcizar depois que o código for implementado.

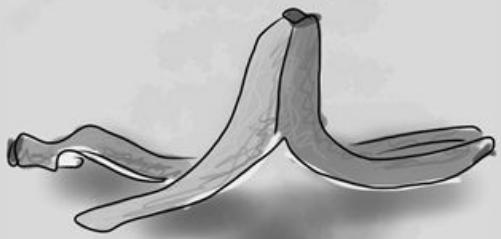
> **PONTO-CHAVE** Considere todos os caminhos em potencial do código *enquanto o estiver escrevendo*. Não planeje tratar os casos “incomuns” mais tarde: você esquecerá e seu código conterá bugs.

Escrever um bom código não tem a ver com ser otimista ou pessimista. Não tem a ver com a quantidade de água no copo nesse momento. Tem a ver com criar um copo tão hermeticamente fechado que a água não se derramará, independentemente da quantidade contida nele.

Perguntas

1. Que tipos de problema você já observou em códigos em que as situações “inesperadas” não eram tratadas adequadamente?
2. Você sempre inclui um tratamento de erros robusto em todos os seus códigos?
3. Quando é aceitável deixar de lado um tratamento rigoroso de erros?

4. Em que outros cenários surpreendentes você pode pensar, que possam afetar a qualidade e a robustez de seu código?

Veja também

- *Não ignore esse erro!* (Capítulo 8) – Conselhos para tratar condições de erro – e uma advertência para *esperar* por elas.
- *Chafurdando na lama* (Capítulo 7) – Considere todas as condições em potencial em seu código, mesmo aquelas que sejam *pouco prováveis*; do contrário, você acabará em um lamaçal confuso e perigoso.
- *Caça aos bugs* (Capítulo 10) – Se todos os casos não forem tratados de forma apropriada, você estará introduzindo bugs como consequência.
- *É hora de testar* (Capítulo 11) – Um bom sistema de testes pode ajudar a enumerar e a monitorar condições inesperadas.

> **TENTE ISTO...** Inspecione a última seção de código em que você trabalhou. Faça uma auditoria no código para ver o nível de zelo no tratamento de erros e as interações incomuns em potencial. Como isso pode ser melhorado?

10.000 MACACOS
(OU ALGO POR AÍ)

CAPÍTULO 10
Caça aos bugs

Se debugging é o processo de remover bugs de software,
a programação deve ser o processo de inseri-los.

– Edsger Dijkstra

Está aberta a temporada: uma temporada que dura o ano todo. Não é preciso ter nenhuma licença; nenhuma restrição será imposta. Pegue uma espingarda e dirija-se aos campos abertos de software para acabar com esses bichos pestilentos – os *bugs* ardilosos –, esmague-os e mate-os.

Tudo bem, a realidade não é tão doce assim. Às vezes, porém, acabamos trabalhando com um código em que podemos jurar que os bugs estão se multiplicando e se juntado contra nós. Uma espingarda é a única resposta.

A história é antiga e diz mais ou menos o seguinte: os programadores escrevem códigos. Os programadores não são perfeitos. O código de um programador não é perfeito. Desse modo, ele não funcionará perfeitamente na primeira vez. Portanto temos bugs.

Se criarmos programadores melhores, certamente criaremos bugs melhores.

Alguns bugs são erros simples, óbvios de serem identificados e fáceis de corrigir. Quando encontramos esses tipos de bug, temos sorte.

A maioria deles – aqueles em que investimos horas de esforço para identificar, perdendo nossos cabelos ou a cor deles nessa busca – são problemas terríveis e sutis. Correspondem às interações incomuns e surpreendentes. São as consequências inesperadas de nossos algoritmos: o comportamento aparentemente não determinístico de um software que parecia ser tão simples. O código só pode ter sido infectado por espíritos

do mal.

Esse não é um problema limitado aos programadores iniciantes, sem muita experiência. Os experts também estão sujeitos a ele. Os pioneiros de nossa arte sofreram; o famoso cientista da computação Maurice Wilkes escreveu: *"Lembro-me muito bem [...] de que, em uma de minhas jornadas entre a sala do EDSAC e o equipamento de perfuração, hesitante nos degraus da escada, percebi claramente que boa parte do restante de minha vida seria passada encontrando erros em meus próprios programas."*[1]

Portanto encare os fatos. Você fará muito debugging. É melhor se acostumar com isso. E é melhor ser bom nisso. (Ao menos você pode se consolar ao pensar que terá muitas oportunidades para praticar.)

Uma preocupação econômica

Quanto tempo você acha que é gasto em debugging? Some os esforços de todos os programadores de todos os países pelo mundo. Vá em frente, dê um palpite.

Impressionantes 312 bilhões de dólares por ano são gastos com custos salariais de programadores depurando seus softwares. Para pôr isso em perspectiva, isso corresponde ao dobro da ajuda financeira feita em toda a zona do euro desde 2008! Esse valor altíssimo, porém realista, é proveniente de pesquisas feitas pela Judge Business School da Universidade de Cambridge.[2]

Você tem uma responsabilidade ao corrigir os bugs mais rapidamente: *salvar a economia global*. A situação mundial está em suas mãos.

Entretanto não são somente os custos salariais. Considere todas as demais implicações de um software com bugs: atrasos nos lançamentos, projetos cancelados, danos à reputação em virtude de um software não confiável e os custos de bugs corrigidos em softwares já lançados.

Um grama de prevenção

Seria um descuido se um capítulo sobre debugging não enfatizasse o quanto é melhor evitar que os bugs, antes de tudo, se manifestassem, em

vez de tentar uma correção após a sua ocorrência. *Um grama de prevenção vale um quilo de cura.* Se o custo do debugging é astronômico, devemos ter como objetivo principal atenuar isso evitando criar bugs em primeiro lugar.

Isso, em um truque editorial clássico, é material para um capítulo diferente, portanto não investigaremos o tema exaustivamente aqui. Lembre-se do conselho nos capítulos 9 e 17.

Basta dizer que devemos sempre empregar técnicas sólidas de engenharia que minimizem a probabilidade de ter surpresas desagradáveis. Um design bem pensado, revisões de código, programação aos pares e uma estratégia de teste bem planejada (incluindo práticas de TDD e suítes de testes de unidade totalmente automatizadas) são da máxima importância. Técnicas como asserções, programação defensiva e ferramentas para cobertura do código ajudarão a minimizar as chances de os erros não serem detectados.

Todos nós conhecemos esses mantras, não é mesmo? Mas o quão diligentes nós somos ao empregar essas táticas?

> **PONTO-CHAVE** Evite injetar bugs em seu código empregando práticas sólidas de engenharia. Não espere que um código rapidamente implementado com hacks tenha boa qualidade.

O melhor conselho para evitar bugs é não escrever um código incrivelmente "inteligente" (que normalmente equivale a um código complexo). Brian Kernighan afirma que *o debugging é duas vezes mais complicado que a escrita do código. Sendo assim, se você escrever o código do modo mais inteligente possível, por definição, você não será esperto o suficiente para depurá-lo.* Martin Fowler nos lembra de que *qualquer tolo pode escrever um código que um computador possa entender. Bons programadores escrevem códigos que os seres humanos possam entender.*

Caça aos bugs

Tome cuidado com os bugs no código anterior. Só provei que ele está correto; não tentei usá-lo.

– Donald Knuth

Sendo realista, não importa o quão sólido seja o seu sistema de implementação de código, alguns desses bugs perniciosos sempre conseguirão passar pelas defesas e exigirão que você vista a capa de caçador do programador e tenha uma espingarda antibugs. Como os encontramos e os eliminamos? Essa pode ser uma tarefa hercúlea, semelhante a encontrar uma agulha em um palheiro. Ou, de modo mais preciso, uma agulha em uma pilha de agulhas.

Encontrar e corrigir um bug é como resolver um quebra-cabeça lógico. Em geral, o problema não será tão difícil se for abordado metodicamente; a maioria dos bugs é facilmente encontrada e corrigida em minutos. Entretanto alguns são piores e exigem mais tempo. A quantidade de bugs difíceis é menor, porém, dada a sua natureza, são neles que gastaremos a maior parte de nosso tempo.

Dois fatores normalmente determinam a dificuldade de corrigir um bug:

- como reproduzi-lo;
- o tempo entre a causa do bug ter sido inserida no código, ou seja, a “falha do software” propriamente dita – a linha de código ruim ou a suposição incorreta na integração – e você realmente percebê-lo.

Se um bug tiver uma pontuação alta em relação a esses dois fatores, será quase impossível encontrá-lo sem ferramentas afiadas e um intelecto aguçado. Há inúmeras técnicas e estratégias práticas que podemos empregar para solucionar o quebra-cabeça e localizar a falha.

A primeira e mais importante tarefa é investigar metodicamente o bug e caracterizá-lo. Dê a si mesmo o melhor material bruto possível para trabalhar:

- Reduza o bug ao conjunto mais simples possível de *passos para reproduzi-lo*. Isso é vital. Separe todos os detalhes externos que não estejam contribuindo com o problema e que servem somente para causar distrações.
- Certifique-se de que você está focando em um único problema. Pode ser muito fácil entrar em um emaranhado quando não percebemos

que estamos misturando duas falhas diferentes – porém relacionadas – em uma só.

- Determine se o problema pode ser repetido. Com que frequência os passos para reproduzi-lo demonstram o problema? Ele se baseia em uma série simples de ações? Depende da configuração do software ou do tipo de computador em que você o está executando? Os dispositivos periféricos associados fazem alguma diferença? Todas essas questões são cruciais no trabalho de investigação a ser feito.

Na verdade, quando você tiver um único conjunto de passos definido para a reprodução do bug, você terá vencido a maior parte da batalha.

As seções a seguir discutem algumas das estratégias mais úteis de debugging.

Coloque armadilhas

Você tem um comportamento errante. Você conhece um ponto em que o sistema parece estar correto; talvez seja na inicialização, mas esperamos que seja bem depois, definido por meio dos passos de reprodução do bug. Você consegue atingir um ponto em que o estado é inválido. Encontre lugares no caminho do código *entre* esses dois pontos e coloque armadilhas para capturar a falha.

Adicione asserções ou testes para verificar as *invariantes* do sistema – os fatos que devem ser mantidos para que o estado esteja correto. Acrescente printouts para diagnósticos de modo a ver o estado do código e descobrir o que está acontecendo.

Ao fazer isso, você terá uma melhor compreensão do código, raciocinará melhor sobre a sua estrutura e, provavelmente, irá adicionar mais asserções à mistura para provar que suas suposições se sustentam. Algumas dessas asserções sobre condições invariantes no código serão genuínas; outras serão relevantes para essa execução em particular. Ambas são ferramentas válidas que irão ajudar você a identificar o bug. Em algum momento, uma armadilha será acionada e o bug será cercado.

> **PONTO-CHAVE** As asserções e a criação de logs (mesmo um humilde `printf`) são ferramentas eficazes de debugging. Use-as com frequência.

Pode valer a pena deixar logs para diagnóstico e asserções no código depois de ter encontrado e corrigido o problema. Porém tome cuidado para não sujar o código com um logging inútil, que oculte o que realmente está acontecendo, gerando um ruído desnecessário no debugging.

Aprenda a fazer divisões binárias

Tenha como meta usar uma estratégia de *divisão binária* para focar nos bugs o mais rapidamente possível.

Em vez de executar passos individuais pelos caminhos do código, descubra o início e o fim de uma cadeia de eventos. Em seguida, particione o espaço do problema em dois e descubra se o ponto central está bom ou ruim. De acordo com essa informação, você terá reduzido o espaço do problema para algo semelhante à metade. Repita isso algumas vezes e logo você terá se acercado do problema.

Essa é uma abordagem bem eficaz – ela permite que você chegue a uma solução em um período de tempo de ordem $O(\log n)$ em vez de $O(n)$. Isso é significativamente mais rápido.

> **PONTO-CHAVE** Faça divisões binárias nos espaços dos problemas para obter resultados mais rápidos.

Utilize essa técnica juntamente com a instalação de armadilhas. Ou com as outras técnicas descritas a seguir.

Empregue a arqueologia de software

A *arqueologia de software* descreve a arte de vasculhar os registros históricos de seu sistema de controle de versões. Isso pode fornecer um caminho excelente para identificar o problema; geralmente, é uma maneira surpreendentemente simples de caçar um bug.

Determine um ponto no passado próximo da base de código em que esse bug não existia. De posse de seu caso de teste reproduzível, avance no tempo para determinar qual conjunto de código provocou a falha. Novamente, uma estratégia de divisão binária é a melhor aposta nesse

caso.[3]

Depois que encontrar a mudança de código que provocou a falha, a causa normalmente será óbvia e a correção será evidente. (Esse é outro motivo convincente para efetuar uma série de check-ins pequenos, frequentes e atômicos em vez de realizar commits massivos que cubram uma variedade de mudanças de uma só vez.)

Teste, teste, teste

À medida que desenvolver o seu software, invista tempo em escrever uma suíte de testes de unidade. Isso irá não apenas ajudar a moldar a maneira como você desenvolve e verifica o código escrito inicialmente, como também atuará como um excelente dispositivo de aviso prévio para alterações que você fizer posteriormente. De modo semelhante ao canário dos mineradores[4], os testes falharão muito antes de o problema se tornar difícil de encontrar e caro para corrigir.

Esses testes também podem atuar como ótimos pontos de partida para iniciar sessões de debugging. Um teste de unidade simples e reproduzível é uma estrutura muito mais fácil de depurar do que um programa completo em execução que deva ser iniciado e ter uma série de ações manuais executadas para que a falha seja reproduzida. Por esse motivo, é aconselhável escrever um teste de unidade que demonstre um bug em vez de começar a caçá-lo a partir de um "sistema completo" em execução.

Depois que você tiver uma suíte de testes, considere empregar uma ferramenta de *cobertura de código* para inspecionar qual parte de seu código realmente está sendo coberta pelos testes. Você poderá ficar surpreso. Uma regra geral simples é esta: se sua suíte de testes não exercitar essa parte, você não poderá acreditar que ela funcione. Mesmo que pareça não haver problemas agora, sem um *test harness*[5], é muito provável que haverá problemas no futuro.

> **PONTO-CHAVE** Um código não testado é um campo de reprodução de bugs. Os testes servem para desinfetá-lo.

Quando a causa de um bug for finalmente determinada, considere escrever um teste simples que ilustre claramente o problema e adicione

esse teste à suíte *antes* de corrigir o código. Isso exige uma verdadeira disciplina, pois, após encontrar o código culpado, você naturalmente irá querer corrigi-lo o mais rápido possível e divulgar a correção. Em vez de fazer isso, crie primeiro um test harness para demonstrar o problema e use-o para provar que ele foi corrigido. O teste servirá para evitar que o bug reapareça no futuro.

Invista em ferramentas afiadas

Há muitas ferramentas com as quais vale a pena se familiarizar, incluindo verificadores de memória como o Electric Fence e ferramentas do tipo canivete suíço como o Valgrind. Vale a pena conhecê-los *agora* em vez de ir atrás deles na última hora. Se souber usar uma ferramenta antes de ter um problema que a exija, você será muito mais eficiente.

Conhecer uma variedade de ferramentas evitará que você quebre uma noz usando um martelo pneumático.

É claro que a ferramenta campeã de debugging é o *debugger*. É o rei das ferramentas, que permite entrar em um programa em execução, avançar uma única instrução ou entrar e sair de funções. Outros recursos muito práticos incluem a capacidade de observar mudanças em variáveis, definir breakpoints condicionais (por exemplo, "break if x > y") e mudar os valores de variáveis durante a execução para testar rapidamente diferentes caminhos no código. Alguns debuggers sofisticados permitem até mesmo que você dê um passo para trás (isso realmente parece ser vodu).

A maioria dos IDEs tem um debugger incluído, portanto definir um breakpoint jamais será uma tarefa inacessível. No entanto você pode achar que vale a pena investir em uma alternativa de mais alta qualidade; não confie na primeira ferramenta que cair em suas mãos.

Em alguns círculos, há um verdadeiro desprezo pelo debugger. *Programadores de verdade não precisam de debugger.* Até certo ponto, isso é verdade; confiar demais no debugger é ruim. Percorrer o código passo a passo, sem pensar muito, pode fazer com que você foque erroneamente no nível micro, em vez de pensar no formato macro e geral do código.

Mas isso não é um sinal de fraqueza. Às vezes, é muito mais fácil e rápido

sacar as armas mais pesadas. Não tenha medo de usar a ferramenta correta para o trabalho.

> **PONTO-CHAVE** Aprenda a usar bem o seu debugger. Use-o nos momentos adequados.

Remova códigos para excluí-lo da análise de causa

Quando puder reproduzir um problema, considere a remoção de tudo que não pareça contribuir com ele para ajudar a focar nas linhas de código ofensoras. Desabilite outras threads que *não deveriam* estar envolvidas. Remova subseções do código que não pareçam estar relacionadas.

É comum descobrir objetos indiretamente relacionados à “área do problema” – por exemplo, por meio de transmissão de mensagem ou de um sistema de notificador-listener. Desconecte fisicamente esse acoplamento (mesmo que você esteja *convencido* de que ele seja benigno). Se o problema ainda puder ser reproduzido, você terá provado que o seu palpite sobre o isolamento estava correto e reduzirá o espaço do problema.

Em seguida, considere remover ou pular seções de código que estejam no caminho do erro (desde que faça sentido do ponto de vista prático). Apague ou remova blocos que não pareçam estar envolvidos comentando-os.

A limpeza evita uma infecção

Não permita que os bugs permaneçam em seu software por mais tempo que o necessário. Não deixe que eles fiquem por lá.

Não menospreze problemas pequenos como sendo *conhecidos*. Essa é uma prática perigosa. Pode levar à *síndrome da janela quebrada*[6], fazendo com que gradualmente pareça normal e aceitável ter um comportamento com bugs. Esse comportamento ruim e persistente pode mascarar as causas de outros bugs que você estiver caçando.

> **PONTO-CHAVE** Corrija os bugs o mais rápido que puder. Não deixe que eles se acumulem até você estar preso em uma fossa de código.

Um projeto em que trabalhei era ruim a ponto de ser desmoralizante a esse respeito. Ao receber um relatório de bug para corrigir, antes de conseguir reproduzir o bug inicial, você encontrava dez problemas diferentes que também deveriam ser corrigidos e que poderiam (ou não) ter contribuído com o bug em questão.

Estratégias oblíquas

Às vezes, você pode bater a cabeça contra um problema complicado durante horas e não chegar a lugar algum. Tente usar uma estratégia oblíqua para evitar ficar preso em um único caminho de debugging.

Faça uma pausa

É importante aprender quando você deve simplesmente parar e se afastar. Uma pausa pode lhe fornecer uma nova perspectiva.

Pode ajudar você a pensar com mais cuidado. Em vez de correr diretamente de volta ao código, faça uma pausa para considerar a descrição do problema e a estrutura do código.

Dê uma volta por aí para se forçar a se afastar de seu teclado. (Quantas vezes você já não teve esses momentos “heureca” no chuveiro? Ou no banheiro?! Acontece comigo o tempo todo.)

Explique o problema para outra pessoa

Descreva o problema para outra pessoa. Com frequência, ao descrever qualquer problema (incluindo a caça a um bug) para outra pessoa, você irá explicá-lo instantaneamente para si mesmo e irá resolvê-lo.

Se não houver uma pessoa de verdade para conversar, você poderá seguir a *estratégia do patinho de borracha* descrita por Andrew Hunt e David Thomas.[7] Converse com um objeto inanimado em sua mesa e explique o problema para si mesmo. Isso será um problema somente se o patinho de borracha começar a responder.

Não se apresse

Depois de encontrar e corrigir um bug, não saia correndo sem pensar. Pare um instante e considere se há outros problemas relacionados à

espreita nessa seção de código. Talvez o problema que você corrigiu seja um padrão que se repita em outras seções do código. Há mais trabalho que poderia ser feito para reforçar o sistema com o conhecimento que você acabou de adquirir?

Tome nota de quais partes do código abrigam mais falhas. Sempre há pontos mais evidentes. Esses pontos correspondem aos 20% de código que 80% dos usuários realmente executam ou a um sinal de software de pouca qualidade, mal escrito. Se você gastar muito tempo tomando notas, pode ser que valha a pena dedicar tempo a essas áreas com problema: talvez sejam necessários uma reescrita, uma revisão profunda de código ou um novo conjunto extra de testes de unidade.

Bugs que não podem ser reproduzidos

Às vezes, você descobre um bug para o qual não é possível criar facilmente um conjunto de passos para reproduzi-lo. O bug desafia a lógica e a razão; não é possível determinar a causa e o efeito. Esses bugs terríveis e intermitentes parecem ser causados por *raios cósmicos* em vez de serem provocados por qualquer interação direta com o usuário. É necessário muito tempo para identificá-los, geralmente porque nunca temos a oportunidade de vê-los em um computador de desenvolvimento com frequência ou quando estamos executando o código em um debugger.

Como fazemos para encontrar e corrigir essas criaturas demoníacas?

- Mantenha registros dos fatores que contribuam com o problema. Com o tempo, você poderá identificar um padrão que ajudará a identificar as causas comuns.
- À medida que obter mais informações, comece a tirar conclusões. Talvez você possa identificar mais dados para manter no registro.
- Considere adicionar mais logs e asserções em versões beta ou oficiais para ajudar na coleta de informações em campo.
- Se for um problema realmente premente, defina um conjunto de testes para execução de *soak tests*[8] de longa duração. Se puder automatizar o comportamento do sistema de maneira representativa, você poderá

agilizar a temporada de caça.

Há alguns aspectos que sabemos que podem contribuir com esses bugs inconstantes. Eles podem fornecer pistas sobre os locais em que podemos começar a investigar:

Código em threads

Como as threads se entrelaçam e interagem de modos não determinísticos e difíceis de serem reproduzidos, com frequência, elas contribuem com as falhas intermitentes estranhas.

Geralmente, esse comportamento é bem diferente quando uma pausa é feita no código usando um debugger, portanto é difícil fazer uma observação forense. O logging também pode alterar a interação entre as threads e mascarar o problema. E builds de "debug" não otimizadas de seu software podem ter um desempenho bem diferente dos builds de "release".

Esses problemas são carinhosamente conhecidos como *Heisenbugs* em homenagem ao "efeito do observador" do físico Werner Heisenberg na mecânica quântica. O ato de observar um sistema pode alterar o seu estado.

Interação com a rede

Por definição, as redes podem se tornar mais lentas e travar a qualquer momento. A maior parte dos códigos presume que todo acesso aos armazenamentos *locais* funciona (porque, na maioria das vezes, é isso que ocorre). É uma falta de cuidado e não deve ser assumido no caso de armazenamentos feitos por meio de uma rede, em que falhas e tempos de carga longos e intermitentes são comuns.

Velocidade variável do armazenamento

Não é somente a latência da rede que pode provocar esse problema. Discos lentos ou operações em bancos de dados podem alterar o comportamento de seu programa, especialmente se você estiver se equilibrando precariamente nos limiares dos timeouts.

Corrupção de memória

Ah, a humanidade! Quando seu código anômalo sobrescrever parte da

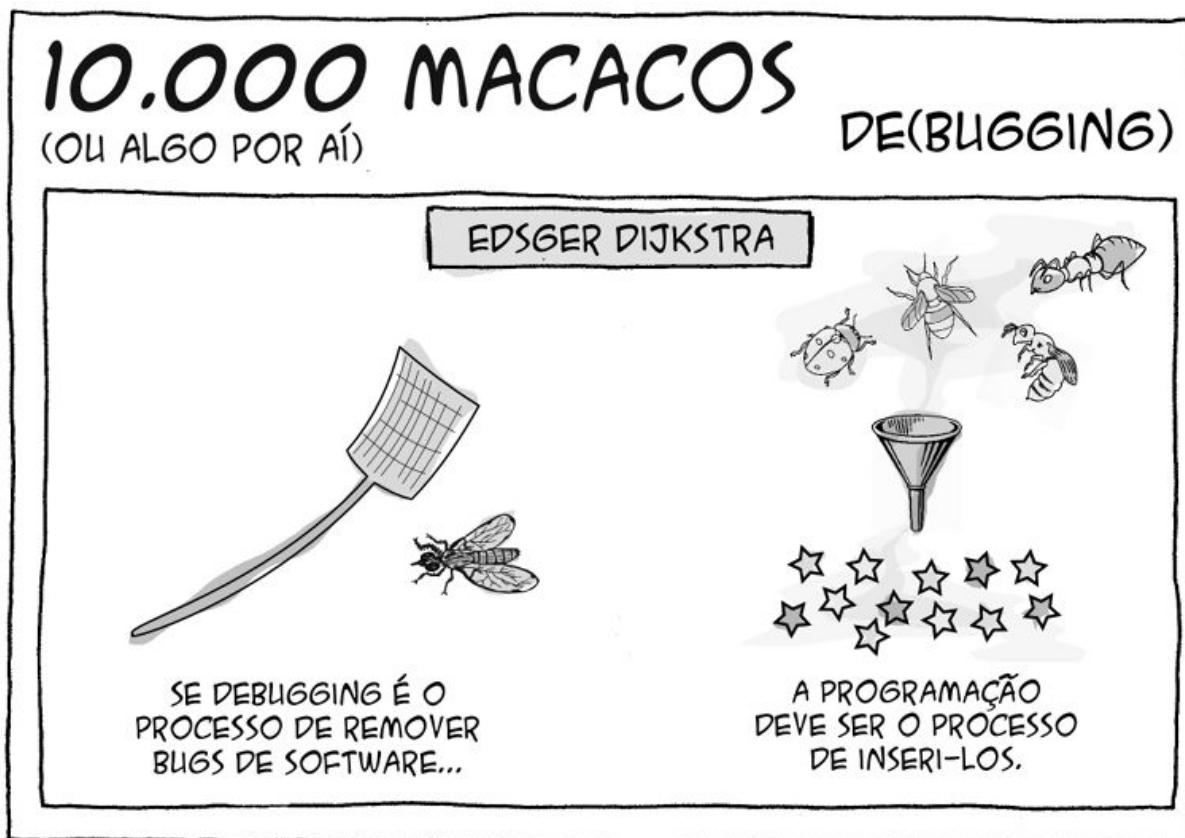
pilha ou da heap, você poderá ver uma variedade de ocorrências estranhas que não poderão ser reproduzidas e serão *muito* difíceis de ser detectadas. A arqueologia de software normalmente é o caminho mais fácil para diagnosticar esses problemas.

Variáveis globais/singletons

Pontos de comunicação fixos no código podem ser os responsáveis por um comportamento imprevisível. Pode ser impossível argumentar sobre a correção de seu código ou prever o que acontecerá se qualquer um a qualquer momento puder acessar um estado global e ajustá-lo puxando o seu tapete.

Conclusão

Depurar o código não é fácil. Mas a culpa é só nossa. Nós criamos os bugs.

Efetuar um debugging eficiente é uma habilidade essencial para qualquer programador.

Perguntas

1. Avalie que parcela de seu tempo você acha que é gasta com debugging. Considere todas as atividades que não estejam relacionadas com a escrita de uma linha de código nova em um sistema.
2. Você gasta mais tempo com debugging em novas linhas de código que você escreveu ou com ajustes em códigos existentes?
3. A existência de uma suíte de testes de unidade para códigos existentes muda a quantidade de tempo que você gasta com debugging ou a maneira de depurar?
4. Ter um software livre de bugs é uma meta realista? É possível alcançar essa meta? Quando é apropriado ter genuinamente como meta um software livre de bugs? O que determina a quantidade ótima de bugs em um produto?

Veja também

- *Espere pelo inesperado* (Capítulo 9) – A maioria dos bugs é causada pelas falhas em levar em conta todas as possíveis condições no fluxo de controle do código.
- *O fantasma de um código do passado* (Capítulo 5) – Descobrir bugs em seu código antigo força você a rever e a avaliar o seu trabalho passado.
- *É hora de testar* (Capítulo 11) – Utilize os testes de unidade para documentar e ajudar a corrigir os bugs encontrados e para evitar futuras regressões no código.

TENTE ISTO... Na próxima vez que você se deparar com um bug, tente usar uma abordagem mais metódica para descobrir a causa. Como é possível empregar a inserção de armadilhas, a divisão binária e as ferramentas afiadas para identificá-lo de modo mais eficiente?

1 Maurice Wilkes, *Memoirs of a Computer Pioneer* (Cambridge, MA: The MIT Press, 1985).

2 "Cambridge University Study States Software Bugs Cost Economy $ 312 Billion per Year" (Estudo da Universidade de Cambridge afirma que os bugs de software custam 312 bilhões de dólares por ano à economia, *http://undo-software.com/company/press/press-release-8*).

3 A ferramenta `git bisect` automatiza essa divisão binária, e vale a pena tê-la em sua caixa de ferramentas se você for usuário do Git.

4 N.T.: Os canários eram levados às minas de carvão para detectar gases tóxicos. Os pássaros seriam afetados por esses gases bem antes dos mineiros, servindo de aviso para um perigo iminente.

5 N.T.: Test harness é o conjunto de softwares e de dados usados para testar uma unidade de programa executando-o em diversas condições e monitorando seu comportamento e os resultados (fonte: *http://en.wikipedia.org/wiki/Test_harness*).

6 A *teoria da janela quebrada* (*http://en.wikipedia.org/wiki/Broken_windows_theory*) afirma que manter a vizinhança em boas condições evita vandalismos e crime.

7 Andrew Hunt e David Thomas, *O programador pragmático* (Bookman, 2010).

8 N.T.: Testes com uma carga significativa realizados durante um período longo de tempo para ver como o sistema se comporta quando houver um uso prolongado (fonte: *http://en.wikipedia.org/wiki/Soak_testing*).

CAPÍTULO 11

É hora de testar

A qualidade é gratuita, mas somente para aqueles dispostos a pagar um preço alto por ela.

– Tom DeMarco e Timothy Lister
Peopleware: como gerenciar equipes e projetos tornando-os mais produtivos

TDD (Test-Driven Development, ou Desenvolvimento orientado a testes): para alguns, é uma religião. Para outros, é a única maneira sensata de desenvolver código. Para outros, ainda, é uma boa ideia que não é possível pôr em prática. E para outros, então, é puro desperdício de esforços.

O que é realmente o TDD?

O TDD é uma técnica importante para criar softwares melhores, embora ainda haja confusão sobre o que significa ser *orientado a testes* e o que é realmente um *teste de unidade*. Vamos explorar isso e descobrir uma abordagem saudável para os testes feitos pelos desenvolvedores para que possamos escrever um código melhor.

Por que testar?

É simples: *devemos* testar o nosso código.

É claro que você deve executar o seu novo programa para ver se ele funciona. Poucos programadores têm confiança suficiente – ou arrogância – para escrever e disponibilizar um código sem testá-lo de *alguma maneira*. Se você vir que atalhos foram tomados, o código raramente funcionará da primeira vez: problemas serão encontrados, seja pela equipe de QA (Quality Assurance, ou Controle de qualidade) ou – pior ainda – quando um cliente usá-lo.

Reduzindo o ciclo de feedback

Para desenvolver um ótimo software, e desenvolvê-lo bem, os programadores precisam receber *feedback*. Devemos receber feedbacks do modo mais rápido e frequente possível. Boas estratégias de teste reduzem o ciclo de feedback para que possamos trabalhar de modo mais eficiente:

- Saber que o nosso código está funcionando quando ele é usado em campo e que ele fornece resultados precisos aos usuários. Se isso não ocorrer, eles reclamarão. Se esse for o nosso único ciclo de feedback, o desenvolvimento de software seria muito lento e muito caro. Podemos fazer algo melhor.
- Para garantir que o software esteja correto *antes* de efetuarmos o seu lançamento, a equipe de QA testa candidatos a versões oficiais. Isso reduz o ciclo de feedback; as respostas voltam mais rapidamente e evitamos cometer erros caros (e embaraçosos) em campo. Mas podemos fazer algo melhor ainda.
- Queremos verificar se nossos novos subsistemas funcionam antes de integrá-los ao projeto. Normalmente, um desenvolvedor irá iniciar a aplicação e executar seu novo código da melhor maneira possível. Alguns códigos podem ser muito inconvenientes para serem testados dessa forma, portanto é possível criar uma aplicação pequena e separada para um test harness que exercite o código. Esses *testes de desenvolvimento* reduzem o ciclo de feedback; descobrimos se o código está funcionando corretamente *enquanto estivermos trabalhando com ele*, e não depois. Mas podemos fazer algo melhor ainda.
- Os subsistemas são constituídos de unidades menores: classes e funções. Se pudermos obter feedback facilmente sobre a correção e a qualidade do código nesse nível, poderemos reduzir o ciclo de feedback novamente. Os testes no nível mais baixo proporcionam os feedbacks mais rápidos possíveis.

Quanto mais breve for o ciclo de feedback, mais rapidamente poderemos fazer as mudanças no design e mais confiança teremos em nosso código. Quanto mais cedo soubermos que há um problema, mais fácil e menos caro será a correção porque nossa mente ainda estará envolvida com o

problema e nos lembraremos do aspecto do código.

> **PONTO-CHAVE** Para melhorar o nosso desenvolvimento de software, devemos ter um feedback rápido e conhecer os problemas assim que eles surgirem. Boas estratégias de teste oferecem ciclos curtos de feedback.

Os testes manuais (sejam realizados por uma equipe de QA ou pelos programadores inspecionando seus próprios códigos) são trabalhosos e lentos. Para ser bem abrangente, vários passos individuais são necessários, os quais devem ser repetidos sempre que um pequeno ajuste for feito no código.

Mas espere um pouco; os computadores não são bons em tarefas trabalhosas e repetitivas? Certamente podemos usar o computador para executar os testes para nós automaticamente. Isso agilizará a execução dos testes e ajudará a reduzir mais ainda o ciclo de feedback.

Os testes automatizados com um ciclo curto de feedback não ajudarão somente no desenvolvimento do código. Depois que você tiver um conjunto de testes, não será preciso jogá-los fora. Reúna-os em um pool de testes e continue executando-os. Dessa maneira, seu código de teste funcionará como um canário em uma mina – ele sinalizará a ocorrência de qualquer problema antes que ele se torne fatal. Se, no futuro, alguém (mesmo que seja você mesmo em um dia ruim) modificar o código e introduzir um comportamento errante (uma *regressão* funcional), o teste indicará isso imediatamente.

Código para testar código

Então o ideal é automatizar o nosso desenvolvimento testando o máximo possível: *trabalhe de modo inteligente, e não do modo mais árduo*. O seu IDE pode evidenciar os erros de sintaxe enquanto você digita – não seria ótimo se ele pudesse mostrar as falhas em testes com a mesma velocidade?

Os computadores podem executar testes de forma rápida e repetitiva, reduzindo o ciclo de feedback. Embora você possa automatizar as aplicações desktop com ferramentas de testes de UI ou usar uma tecnologia baseada em navegadores, com muita frequência, os testes de desenvolvimento exigem que o programador implemente uma estrutura

de testes que acione o seu código de produção [o SUT (*System Under Test*, ou Sistema em teste)], estimulando-o de determinadas maneiras para verificar se ele responde conforme esperado.

Escrevemos códigos para testar códigos. É bem "meta".

Sim, escrever esses testes exige um tempo precioso do programador. E, sim, sua confiança no código será somente tão boa quanto a qualidade dos testes que você escrever. Porém não é difícil adotar uma estratégia de testes que melhore a qualidade de seu código e torne-o mais seguro. Isso ajuda a *reduzir* o tempo necessário para desenvolver códigos: *mais pressa, menos velocidade*. Estudos mostram que uma estratégia sólida de testes reduz substancialmente a incidência de defeitos.[1]

É verdade que uma suíte de testes pode reduzir a velocidade do desenvolvimento se você escrever testes frágeis, difíceis de serem entendidos, e se o seu código for tão rígido que uma mudança em um método force a reescrita de um milhão de testes. Esse é um argumento contra suítes de teste *ruins*, e não contra os testes em geral (da mesma maneira que um código ruim não é um argumento contra a programação em geral).

Quem escreve os testes?

No passado, algumas pessoas defendiam uma função de "engenheiro de testes de unidade" que fosse especializado em verificar o código de um programador que tivesse feito um trabalho antes. Porém a abordagem mais eficiente é que os próprios programadores escrevam seus testes de desenvolvimento.

Afinal de contas, você estará testando o seu código à medida que implementá-lo, de qualquer modo.

PONTO-CHAVE Precisamos de testes em todos os níveis da pilha de software e do processo de desenvolvimento. Entretanto os programadores, particularmente, exigem testes no escopo mais baixo possível para reduzir o ciclo de feedback e ajudar no desenvolvimento de um software com mais qualidade da forma mais rápida e fácil possível.

Tipos de teste

Há vários tipos de teste e, com frequência, quando você ouvir alguém falar de "teste de unidade", essas pessoas provavelmente irão se referir a algum outro tipo de teste de código. Nós empregamos os termos:

Testes de unidade

Os testes de unidade exercitam especificamente as menores "unidades" funcionais *isoladamente* para garantir que cada função esteja correta. Se o teste não envolver uma única unidade de código (que *pode* ser uma classe ou uma função) isoladamente (ou seja, sem envolver outras "unidades" do código de produção), ele não será um teste de unidade.

Esse isolamento significa especificamente que um teste de unidade não envolverá nenhum acesso externo: nenhum banco de dados, nenhuma rede ou operações no sistema de arquivos serão executados.

O código do teste de unidade normalmente é escrito usando um framework de estilo "xUnit" pronto. Toda linguagem e todo ambiente têm um conjunto deles e outros têm um padrão consolidado pelo uso. Não há nada mágico em um framework de teste e você pode ir bem longe ao escrever testes de unidade usando apenas um simples `assert`. Daremos uma olhada nos frameworks mais adiante.

Testes de integração

Esses testes inspecionam o modo como as unidades individuais se integram em conjuntos maiores e coesos de funcionalidades em cooperação. Verificamos se os componentes integrados se encaixam e operam corretamente em conjunto.

Os testes de integração geralmente são escritos nos mesmos frameworks dos testes de unidade; a diferença está somente no escopo do sistema em teste. Os "testes de unidade" de muitas pessoas são realmente testes de nível de integração, que lidam com mais de um objeto no SUT. Para dizer a verdade, o nome que damos a esse teste nem chega perto da importância do fato de o teste existir!

Testes de sistema

Também conhecido como testes *fim a fim* (end-to-end), esses testes podem ser vistos como uma especificação da funcionalidade exigida do sistema todo. Eles são executados em relação à pilha totalmente integrada de software e podem ser usados como critérios de aceitação do projeto.

Os testes de sistema podem ser implementados como um código que exercite as APIs públicas e os pontos de entrada do sistema, ou podem guiar o sistema de fora usando uma ferramenta como o Selenium – uma ferramenta para automatizar um navegador web. Pode ser difícil testar todas as funcionalidades de uma aplicação de modo realista por meio de sua camada de UI, caso em que empregamos *testes subcutâneos* que direcionam o código a partir da camada imediatamente abaixo da lógica da interface.

Por causa do escopo mais amplo dos testes de sistema, a suíte completa de testes pode exigir um tempo considerável para ser executada. Pode haver muito tráfego de rede envolvido ou acessos lentos ao banco de dados a serem levados em consideração. Os custos de configuração e desmontagem do ambiente podem ser enormes para ter o SUT pronto para executar os testes de sistema.

Cada nível de teste dos desenvolvedores estabelece vários fatos sobre o SUT e compõe uma série de *casos de teste* que provam que esses fatos se sustentam.

Há estilos diferentes de desenvolvimento orientado a testes. Um projeto pode ser orientado por uma mentalidade voltada aos testes de unidade, em que você espera ver mais testes de unidade do que de integração, e mais testes de integração do que de sistema. Ou pode ser orientado por uma mentalidade voltada a testes de sistema, que é o inverso, com muito menos testes de unidade. Cada tipo de teste é importante por si só, e todos devem estar presentes em um projeto de software maduro.

Quando os testes devem ser escritos

O termo TDD (que quer dizer *Test-Driven Development*, ou Desenvolvimento orientado a testes) implica em um desenvolvimento com

testes antes, embora, na realidade, haja dois temas separados nesse caso. Você pode "orientar" o seu design pelo feedback dado pelos testes sem escrever religiosamente esses testes antes.

No entanto, quanto mais tempo você levar para escrever seus testes, menos eficientes eles serão: você se esquecerá de como o código deve funcionar, falhará em cuidar dos casos limítrofes ou, quem sabe, esquecerá totalmente de escrever os testes. Quanto mais tempo você levar para escrever os seus testes, mais lento e menos eficiente será o seu ciclo de feedback.

A abordagem "TDD" de testes antes é comumente vista em círculos XP (Extreme Programming). O mantra é: *não escreva nenhum código de produção a menos que você tenha um teste em falha*. O ciclo TDD de testes antes consiste em:

1. Determinar a próxima funcionalidade necessária. Escreva um teste para a sua nova funcionalidade. É óbvio que ele irá falhar.
2. Somente então implemente essa funcionalidade da maneira mais simples possível. Você sabe que sua funcionalmente estará pronta quando o teste passar. À medida que codificar, você poderá executar a suíte de testes várias vezes. Como cada passo adiciona uma pequena parte da funcionalidade e, sendo assim, um teste pequeno, esses testes deverão ser executados rapidamente.
3. *Esta é uma parte importante que normalmente é menosprezada*: organize o código. Refatore partes comuns que não estejam adequadas. Reestruture o SUT para ter uma estrutura interna melhor. Você pode fazer tudo isso com toda a confiança de que nada deixará de funcionar, pois haverá uma suíte de testes em relação à qual você poderá fazer a validação.
4. Volte ao passo 1 e repita até ter escrito casos de teste que sejam aprovados para todas as funcionalidades necessárias.

Esse é um ótimo exemplo de um ciclo de feedback eficaz e gloriosamente curto. Geralmente, ele é referenciado como o ciclo *vermelho-verde-refatorar* (red-green-refactor) em homenagem às ferramentas de testes de

unidade que mostram os testes em falha como uma barra de progresso vermelha e os testes que passarem como uma barra verde.

Mesmo que você não honre o mantra do "teste antes", mantenha o seu ciclo de feedback curto e escreva testes de unidade durante ou imediatamente depois de uma seção de código. Os testes de unidade realmente ajudam a "orientar" o nosso design: eles não só garantem que tudo esteja funcionalmente correto e evitam regressões como também são uma ótima maneira de explorar como uma API de classe será usada em produção – sua facilidade de uso e sua organização. Esse é um feedback de valor inestimável. Os testes também se destacam como uma documentação útil sobre o uso de uma classe depois que ela estiver completa.

> **PONTO-CHAVE** Escreva testes *à medida que escrever* o código em teste. Não adie a escrita dos testes; do contrário, seus testes não serão tão eficientes.

Essa abordagem de testar cedo e com frequência pode ser aplicada nos níveis de unidade, de integração e de sistema. Mesmo que o seu projeto não tenha nenhuma infraestrutura para realizar testes automatizados de sistema, você poderá, ainda assim, assumir a responsabilidade e verificar as linhas de código que você escrever usando testes de unidade. O custo é baixo e, se a estrutura do código for boa, será fácil.[2]

Outro momento essencial para escrever um teste é quando houver necessidade de corrigir um bug no código de produção. Em vez de implementar uma correção rápida no código, escreva antes um teste de unidade em falha que mostre a causa do bug. Às vezes, o ato de escrever esse teste servirá para mostrar outras falhas relacionadas no código. Em seguida, aplique a sua correção do bug e faça o teste passar. O teste deve ser adicionado ao seu pool de testes e servirá para garantir que o bug não reapareça no futuro.

Quando executar os testes

Você pode ver muita coisa somente olhando.

– Yogi Berra

É claro que, se o desenvolvimento for feito usando TDD, você estará executando seus testes *enquanto* estiver desenvolvendo cada funcionalidade para provar que sua implementação estará correta e que será suficiente.

Porém essa não é a única vida que seu código de teste terá.

Adicione tanto o código de produção *quanto* o seu teste no sistema de controle de versões. O teste não será jogado fora; ele será incluído na suíte de testes existente. O teste permanecerá vivo para garantir que seu software continue a funcionar conforme esperado. Se alguém, mais tarde, modificar o código de forma inadequada, a pessoa será alertada do fato antes de conseguir ir muito longe.

Todos os testes devem ser executados no servidor de build como parte do conjunto de ferramentas de *integração contínua*. Os testes de unidade devem ser executados pelos desenvolvedores com frequência em seus computadores de desenvolvimento. Alguns ambientes de desenvolvimento oferecem atalhos para disparar facilmente os testes de unidade; outros sistemas verificam o seu sistema de arquivos e executam os testes de unidade quando os arquivos forem modificados. Entretanto prefiro incluir os testes diretamente no processo de build/compilação/execução. Se minha suíte de testes de unidade falhar, a compilação do código será considerada como *em falha* e o software não poderá ser executado. Dessa maneira, os testes não poderão ser ignorados. Eles serão executados *sempre* que o build do código for feito. Se forem acionados manualmente, os desenvolvedores poderão se esquecer de executar os testes ou irão "evitar a inconveniência" enquanto trabalham.

Injetar os testes diretamente no processo de build também ajuda a manter os testes pequenos e rápidos para executar.

> **PONTO-CHAVE** Incentive o fato de os testes serem executados cedo e com frequência. Inclua-os em seu processo de build.

Os testes de integração e de sistema podem levar tempo demais para executar no computador de um desenvolvedor a cada compilação. Nesse caso, de forma justificável, eles podem ser executados somente no servidor

de builds de CI.

Lembre-se de que os testes automatizados de nível de código não eliminam a necessidade de uma revisão humana de QA antes de o software ser disponibilizado em uma versão. Testes exploratórios feitos por verdadeiros especialistas em teste têm valor inestimável, independentemente da quantidade de testes de unidade, de integração e de sistema criada. Uma suíte de testes automatizada evita a introdução daqueles erros que podem ser facilmente corrigidos e previstos, fazendo com que a equipe de QA não perca tempo. Isso significa que aquilo que for encontrado pelo pessoal de QA serão *realmente* bugs terríveis, e não apenas erros simples. Viva!

> **PONTO-CHAVE** Bons testes de desenvolvimento não substituem testes completos de QA.

O que deve ser testado

Teste tudo o que for importante em sua aplicação. Quais são seus requisitos?

Naturalmente, seus testes devem verificar se toda unidade de código se comporta conforme for necessário, retornando resultados precisos. No entanto, se o desempenho for um requisito importante de sua aplicação, você deve criar testes para monitorar o desempenho do código. Se o seu servidor deve responder a queries em um determinado intervalo de tempo, inclua testes para essa condição.

Pode ser que você queira levar em consideração o nível de *cobertura* de seu código de produção pelos casos de teste. Você pode executar ferramentas que determinem isso. Contudo essa tende a ser uma métrica terrível para ter como meta. Escrever um código de teste que tente arduamente cobrir todas as linhas de código de produção pode representar uma distração enorme; é mais importante focar nos comportamentos e nas características mais relevantes do sistema.

Bons testes

Escrever bons testes exige prática e experiência; é perfeitamente possível escrever testes ruins. Não se preocupe demais com isso no início – é mais importante *começar* realmente a escrever os testes do que ficar paralisado, com medo de que seus testes sejam ruins. Comece a escrever os testes e você começará a aprender.

Testes ruins tornam-se um peso: uma penalidade, em vez de serem um patrimônio. Esses tipos de teste podem reduzir a velocidade do desenvolvimento de código se demorarem muito para serem executados. Podem tornar a modificação do código difícil se uma simples alteração fizer com que muitos testes difíceis de ler falhem.

Quanto mais tempo seus testes levarem para executar, menor será a frequência com que você os executará, menos usados eles serão e menos feedback serão obtidos a partir deles. Eles acrescentarão menos valor.

Certa vez, herdei uma base de código que tinha uma suíte enorme de testes de unidade; parecia ser um ótimo sinal. Infelizmente, esses testes eram um *código legado* pior do que o código de produção. Qualquer modificação que fizéssemos no código causava várias falhas em métodos de teste com centenas de linhas que eram intratáveis, densas e difíceis de entender. Felizmente, essa não é uma experiência comum.

PONTO-CHAVE Testes ruins podem ser uma penalidade. Eles podem impedir um desenvolvimento eficiente.

Estas são as características de um bom teste:

- É conciso e tem um nome claro, de modo que, quando falhar, você poderá facilmente determinar qual é o problema (por exemplo, *lista nova está vazia*).
- É fácil de manter; é fácil de ser escrito, lido e modificado.
- Executa rapidamente.
- Está atualizado.
- Executa sem qualquer configuração prévia do computador (por exemplo, não é necessário preparar os paths de seu sistema de arquivos nem configurar um banco de dados antes de executá-lo).
- Não depende de nenhum outro teste que deva ser executado antes ou

depois dele; não há nenhuma dependência de estados externos nem de qualquer variável compartilhada no código.

- Testa o código de produção *propriamente dito* (Já vi "testes de unidade" que trabalhavam com uma *cópia* do código de produção – uma cópia que estava desatualizada. Não é produtivo. Também já vi comportamentos especiais de "teste" adicionados ao SUT em builds de teste; isso também não é um teste do verdadeiro código de produção.).

Há algumas descrições comuns de testes criados de forma inadequada:

- testes que às vezes executam, às vezes falham (com frequência, isso é causado pelo uso de threads ou de códigos concorrentes que dependem de tempos específicos, dependências externas, a ordem em que os testes são executados na suíte de testes ou estados compartilhados);
- testes com uma aparência horrível, difíceis de ler ou de modificar;
- testes longos demais (testes longos são difíceis de entender e o SUT claramente não será bem isolável se exigir que centenas de linhas devam ser criadas);
- testes que exercitem mais de um aspecto em um único caso de teste (um "caso de teste" é *singular*);
- testes que ataquem uma API de classe, função por função, em vez de estarem voltados para comportamentos individuais;
- testes de códigos de terceiros que não foram escritos por você (não há necessidade de fazer isso, a menos que haja bom motivo para desconfiar deles);
- testes que não cubram realmente a principal funcionalidade ou o comportamento de uma classe, mas ocultam esse fato por trás de uma grande quantidade de testes de aspectos menos importantes (se isso puder ser feito, é porque sua classe provavelmente estará grande demais);
- testes que cubram aspectos sem sentido em detalhes excruciantes (por exemplo, getters e setters de propriedades);

- testes que dependam de conhecimento de "caixa-branca" dos detalhes de implementação internos do SUT (isso significa que você não poderá alterar a implementação sem alterar todos os testes);
- testes que funcionem somente em um computador.

Às vezes, um teste que não cheire bem indica não (somente) um teste ruim, mas um código ruim em teste. Esses cheiros devem ser observados e usados para orientar o design de seu código.

Como é a aparência de um teste?

O framework de teste que você usar determinará o formato de seu código de teste. Ele poderá proporcionar uma configuração estruturada e recursos para desmontá-la, além de uma maneira de agrupar testes individuais em *fixtures* maiores.

Convencionalmente, para cada teste, haverá algum preparativo; em seguida, você realizará uma operação e, por fim, validará o resultado dessa operação. Isso é comumente conhecido como o padrão *arrange-act-assert* (preparar-agir-verificar). Para os testes de unidade, na fase de verificação, normalmente temos uma única verificação como meta – se for preciso escrever várias asserções, seu teste poderá não estar executando um único caso de teste.

Aqui está um exemplo de um método para teste de unidade em Java que segue esse padrão:

```
@Test
public void stringsCanBeCapitalised()
{
    String input = "This string should be uppercase."; ❶
    String expected = "THIS STRING SHOULD BE UPPERCASE.";

    String result = input.toUpperCase(); ❷

    assertEquals(result, expected); ❸
}
```

❶ arrange (preparar) – preparamos o dado de entrada;

❷ act (agir) – realizamos a operação;

❸ assert (verificar) – validamos o resultado dessa operação.

Manter esse padrão ajuda a deixar os testes focados e legíveis.

É claro que esse teste sozinho não cobrirá todas as maneiras em potencial de usar e abusar da passagem de Strings para letras maiúsculas. Precisamos ter mais testes que incluam outros dados de entrada e expectativas. Cada teste deve ser adicionado como um novo método de teste, e não inserido nesse teste.

Nomes de testes

Testes focados têm nomes bem claros que são lidos como frases simples. Se não for possível nomear facilmente um caso de teste, provavelmente seu requisito será ambíguo ou você estará tentando testar vários aspectos.

O fato de o método de teste *ser* um teste normalmente estará implícito (por causa de um atributo como `@Test` que vimos anteriormente), portanto não será preciso acrescentar a palavra `test` ao nome. O exemplo anterior não precisa se chamar `testThatStringsCanBeCapitalised`.

Pense que seus testes serão lidos como especificações de seu código; cada nome de teste é uma afirmação sobre o que o SUT faz – um único fato. Evite palavras ambíguas ou que não acrescentem nenhum valor como “deve”. Assim como ocorre quando criamos nomes em nosso código de produção, evite redundâncias e nomes longos desnecessários.

Os nomes dos testes não precisam seguir as mesmas convenções de estilo do código de produção; eles formam sua própria linguagem específica para o domínio. É comum ver nomes de métodos muito mais longos e o uso liberal de underscores, mesmo em linguagens como C# e Java, em que eles não são idiomáticos (o argumento é que `strings_can_be_capitalised` exige menos esforço para ser lido).

A estrutura dos testes

Certifique-se de que sua suíte de testes cobrirá as funcionalidades importantes de seu código. Considere os casos de entradas “normais”. Considere também os “casos de falha” comuns. Considere o que acontece com valores no limite, incluindo vazios ou zeros. Ter como meta cobrir

todos os requisitos e as funcionalidades de todo o seu sistema com testes de sistema e de integração e cobrir todo o código com testes de unidade é um objetivo louvável. No entanto isso pode exigir esforços sérios.

Não duplique os testes: isso aumenta os esforços, a confusão e o custo de manutenção. Cada caso de teste que você escrever deve verificar um fato; esse fato não precisa ser verificado novamente, seja em um segundo teste ou como parte do teste de outro item. Se o seu primeiro caso de teste verificar uma pré-condição após a construção de um objeto, você poderá supor que essa pré-condição será mantida em todos os demais casos de teste que você escrever – não há necessidade de reproduzir a verificação sempre que um objeto for construído.

Um erro comum é ver uma classe com cinco métodos e achar que serão necessários cinco testes, um para cada método. Essa é uma abordagem compreensível (porém ingênua). Testes baseados em funções raramente são úteis, pois, em geral, você não poderá testar um único método isoladamente. Depois de chamá-lo, você precisará usar outros métodos para inspecionar o estado do objeto.

Em vez de fazer isso, escreva testes que considerem os comportamentos específicos do código. Isso resulta em um conjunto de testes muito mais claro e coeso.

Faça a manutenção dos testes

O seu código de teste é tão importante quanto o código de produção, portanto considere o seu formato e a sua estrutura.

Se o código se tornar confuso, limpe-o e refatore-o. Se o comportamento de uma classe for alterado de modo que o seu teste falhe, não comente simplesmente os testes e fuja. Faça a manutenção dos testes. Pode ser tentador "economizar tempo" quando os prazos estiverem prestes a se esgotar pulando a limpeza do código de testes. Porém, se você for apressado e descuidado nesse caso, isso poderá voltar-se *contra* você.

Em um projeto, recebi um email de um colega: *eu estava trabalhando em sua classe XYZ e os testes de unidade pararam de funcionar, portanto tive de remover todos eles*. Fiquei bem surpreso com isso e dei uma olhada em

quais testes foram removidos. Infelizmente, eram casos de teste importantes que indicavam claramente um problema fundamental com o novo código. Desse modo, restaurei o código de teste e "corrigi" o bug, apoiando a alteração. Em seguida, trabalhamos juntos para compor um novo caso de teste para a funcionalidade necessária e reimplementamos uma versão que satisfizesse os testes antigos e o novo.

> **PONTO-CHAVE** Faça a manutenção de sua suíte de testes e ouça-a quando ela falar com você.

Selecionando um framework de teste

O framework para testes de unidade e de integração que você usar moldará os seus testes, determinando o estilo das asserções e das verificações que poderá ser usado e a estrutura de seu código de teste (por exemplo, os casos de teste serão escritos em funções livres ou como métodos de uma classe fixture de teste?).

Portanto é importante selecionar um bom framework de testes de unidade. Ele não precisa ser complexo nem pesado. Na verdade, é preferível não selecionar uma ferramenta difícil de manusear. Lembre-se de que você pode ir bem longe com um simples `assert`. Geralmente, eu começo a testar um novo protótipo de código somente com um método `main` e uma série de `asserts`.

A maioria dos frameworks de teste segue o modelo "xUnit", que teve origem no Smalltalk SUnit de Kent Beck. Esse modelo foi portado e se popularizou com o JUnit (para Java), embora haja implementações amplamente equivalentes na maioria das linguagens – por exemplo, NUnit (C#) e CppUnit (C++). Esse tipo de framework nem sempre é ideal; o estilo de teste xUnit resulta em código não idiomático em algumas linguagens (em C++, por exemplo, o código é bem desajeitado e anacrônico; outros frameworks de teste podem funcionar melhor – dê uma olhada no *Catch* como uma ótima alternativa[3]).

Alguns frameworks oferecem GUIs elegantes, com barras vermelhas e verdes para indicar sucesso ou fracasso claramente. Isso pode fazê-lo feliz, porém não sou grande fã disso. Acho que você não deveria ter uma UI

separada ou um passo de execução diferente para os testes de desenvolvimento. O ideal é que eles estejam incluídos diretamente em seu sistema de build. O feedback deve ser apresentado instantaneamente, como qualquer outro erro de código.

Os testes de sistema tendem a usar uma forma diferente de framework, em que vemos o uso de ferramentas como o Fit (*http://fit.c2.com/*) e o Cucumber (*http://cukes.info/*). Essas ferramentas tentam definir os testes de uma maneira mais humana e menos programática, permitindo aos que não sejam programadores participar do processo de teste/especificação.

Nenhum código é uma ilha

Quando escrevemos testes de unidade, temos como meta definir unidades de código realmente *isoladas* no "sistema em teste". Essas unidades podem ser instanciadas sem que o restante do sistema esteja presente.

A interação de uma unidade com o mundo externo é expressa por meio de dois contratos: a interface que ela fornece e a interface que ela espera. A unidade não deve depender de mais nada – especificamente, ela não deve depender de nenhum estado global compartilhado nem de objetos singleton.

> **PONTO-CHAVE** Variáveis globais e objetos singleton são um anátema do teste confiável. Não é possível testar facilmente uma unidade que tenha dependências ocultas.

A interface que uma unidade de código *disponibiliza* corresponde simplesmente aos métodos, às funções, aos eventos e às propriedades de sua API. Talvez ela também disponibilize algum tipo de interface de callback.

As interfaces que ela *espera* são determinadas pelos objetos com os quais ela colabora por meio de sua API. São os tipos de parâmetro de seus métodos públicos ou qualquer mensagem que ela possa receber. Por exemplo, uma classe `Invoice` que exija um parâmetro `Date` depende da interface de data.

Os objetos com os quais uma classe colabora devem ser passados como

parâmetros do construtor – uma prática conhecida como *parametrizar de cima*. Isso permite que sua classe evite dependências internas de outros códigos; em vez disso, a ligação é configurada pelo seu proprietário. Se os colaboradores forem descritos por uma *interface* em vez de o serem por um tipo concreto, temos uma abertura pela qual podemos realizar nossos testes; temos a capacidade de prover implementações alternativas de teste.

Esse é um exemplo de como os testes tendem a resultar em um código mais bem fatorado. Eles forçam o seu código a ter menos conexões fixas no código e menos suposições internas. Depender de uma interface mínima que descreva uma colaboração específica também é uma boa prática, em vez de depender de uma classe completa que possa disponibilizar muito mais do que a interface simples exigida.

> **PONTO-CHAVE** Fatorar o seu código para torná-lo "testável" resulta em um melhor design do código.

Ao testar um objeto que dependa de uma interface externa, você pode disponibilizar uma versão "dummy" dessa interface no caso de teste. Os termos variam nos círculos de teste, porém, com frequência, eles são chamados de *dublês de teste* (test doubles). Há diversas formas de dublês, porém estes são os mais comumente usados:

Dummies

Objetos dummies normalmente são estruturas vazias – o teste não os chamará, porém eles existem para satisfazer às listas de parâmetros.

Stubs

Os objetos stub são implementações simplistas de uma interface; normalmente retornam uma resposta predefinida e também podem registrar informações sobre as chamadas feitas a eles.

Mocks

Objetos mock são os reis da terra dos dublês de teste – um recurso oferecido por várias bibliotecas diferentes de simulação. Um objeto mock pode ser criado automaticamente a partir de uma interface nomeada e, em seguida, pode receber previamente informações sobre como o SUT irá utilizá-lo. Uma operação de teste em um SUT é

realizada e, em seguida, você poderá inspecionar o objeto mock para verificar se o comportamento está de acordo com o esperado.

Linguagens diferentes têm diferentes suportes para frameworks de mocking. É mais fácil sintetizar mocks em linguagens com reflexão[4].

O uso sensato de objetos mock pode tornar os testes mais simples e mais claros. É claro, porém, que algo bom pode ser usado em excesso. Os testes repletos de usos complexos de vários objetos mock podem se tornar muito complicados de entender e difíceis de manter. A *mock mania* é outro indício de um código de teste que não cheira bem e pode evidenciar que a estrutura do SUT não esteja correta.

Conclusão

Se você não se importar com a qualidade, qualquer requisito poderá ser atendido.

– Gerald M. Weinberg

Os testes nos ajudam a escrever o nosso código. Eles nos ajudam a escrever um *bom* código. Ajudam a manter a *qualidade* de nosso código. Podem *orientar* o design do código e servir como documentação sobre como usá-lo. Porém os testes não resolvem todos os problemas do desenvolvimento de software. Edsger Dijkstra disse que *os testes de programas podem ser usados para mostrar a presença de bugs, mas nunca para mostrar a sua ausência*.

Nenhum teste é perfeito, porém a existência de testes serve para aumentar a confiança no código que você escrever e no código que você mantiver. O esforço dedicado aos testes de desenvolvimento é um compromisso: que nível de esforço você quer investir em escrever testes para adquirir confiança? Lembre-se de que a sua suíte de testes é somente tão boa quanto os testes nela contidos. É perfeitamente possível deixar passar um caso importante; você pode implantar o sistema em produção e ainda assim deixar um problema passar. Por esse motivo, o código de teste deve ser revisado tão cuidadosamente quanto o código de produção.

Apesar disso, a conclusão é simples: se o código for importante o suficiente para ser escrito, ele será importante o suficiente para ser

testado. Portanto escreva testes de desenvolvimento para o seu código de produção. Use-os para *orientar* o design de seu código. Escreva os testes *enquanto* estiver implementando o código de produção. E automatize a execução desses testes.

Reduza o ciclo de feedback.

Testar é fundamental e importante. Este capítulo somente tocou a superfície do assunto, incentivando-o a testar e estimulando-o a conhecer mais sobre as boas técnicas de testes.

Perguntas

1. A quantos estilos de teste você já foi exposto?
2. Qual é a melhor técnica para testes de desenvolvimento: testar antes ou testar (imediatamente após) a codificação? Por quê? Como a sua experiência moldou essa resposta?
3. Empregar um engenheiro especializado em escrever testes de unidade para ajudar a compor uma suíte de testes de alta qualidade é uma boa ideia?
4. Por que os departamentos de QA normalmente não escrevem muito código de teste e, em geral, focam na execução de scripts de teste e na realização de testes exploratórios?
5. Qual é a melhor maneira de introduzir um desenvolvimento orientado a testes em uma base de código em que testes automatizados nunca foram usados? Que tipos de problema você irá encontrar?
6. Investigue o *desenvolvimento orientado a comportamento* (behaviour-driven development). Como ele difere de um TDD “tradicional”? Quais problemas ele resolve? Ele complementa ou substitui o TDD? É uma direção para a qual você deve voltar os seus testes?

Veja também

- *Marcando um gol* (Capítulo 21) – Os testes dos programadores aumentam a confiança nas versões disponibilizadas à equipe de QA para testes.
- *Mantendo as aparências* (Capítulo 2) – Um bom layout de código e

uma boa apresentação são tão importantes no código de teste quanto no código de produção.

- *Espero pelo inesperado* (Capítulo 9) – É sensato incluir casos de teste para os cenários “menos prováveis”, que seu código poderá não estar esperando.
- *Caça aos bugs* (Capítulo 10) – Use os testes para orientar o seu processo de debugging.
- *Um conto de dois sistemas* (Capítulo 13) – Um exemplo de como os testes de unidade ajudam a melhorar a qualidade do código.

> **TENTE ISTO...** Se você ainda não o fez, comece a escrever testes de unidade para o seu código hoje. Se você já usa testes, preste atenção em como eles servem de base e orientam o design de seu código.

---

1 David Janzen e Hossein Saiedian, “Test-Driven Development Concepts, Taxonomy, and Future Direction” (Conceitos, taxonomia e a futura direção do desenvolvimento orientado a testes), *Computer* 38:9 (2005).

2 Se não houver uma boa estrutura de código, uma tentativa de escrever um teste ajudará a

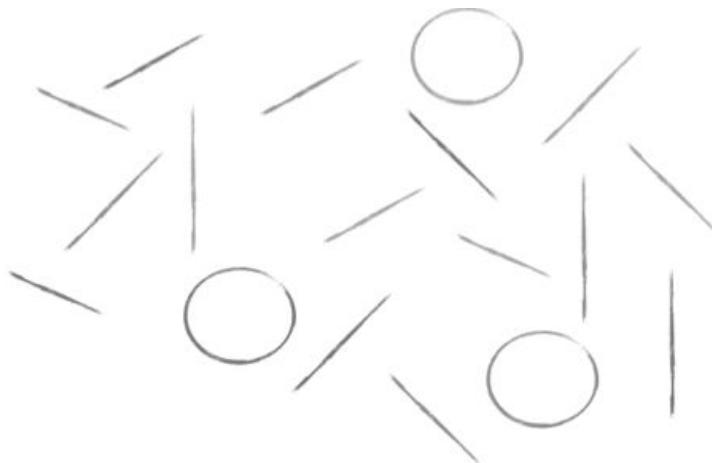
conduzir você em direção a uma estrutura melhor.

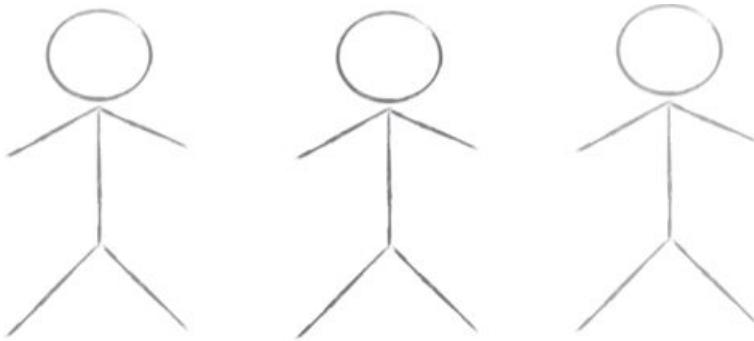
3 O framework de teste de unidade Catch (disponível no GitHub em *https://github.com/philsquared/Catch*).

4 N.T.: Reflexão é a capacidade de um programa observar ou até mesmo de modificar a sua estrutura ou o seu comportamento (fonte: *http://pt.wikipedia.org/wiki/Reflexão_ (programação)*).

CAPÍTULO 12

Lidando com a complexidade

A simplicidade é uma grande virtude, porém exige trabalho árduo para ser alcançada e educação para ser apreciada. E, para piorar, a complexidade vende mais.

– Edsger Dijkstra

O código é complexo. A complexidade é uma batalha contra a qual todos nós devemos lutar diariamente.

É claro que o seu código é ótimo, não é mesmo? É o código das outras pessoas que é complexo.

Bem, não. Nem sempre. Admita. É muito fácil escrever algo complicado. Isso acontece quando você não está prestando atenção. Acontece quando você não planeja com antecedência suficiente. Acontece quando você começa a trabalhar em um problema “simples”, mas logo descobre tantos casos limítrofes que o seu algoritmo simples cresce a ponto de parecer um labirinto, pronto para aprisionar um programador desavisado.

Minha observação é que a complexidade do software tem origem em três fontes principais. Círculos. E linhas.

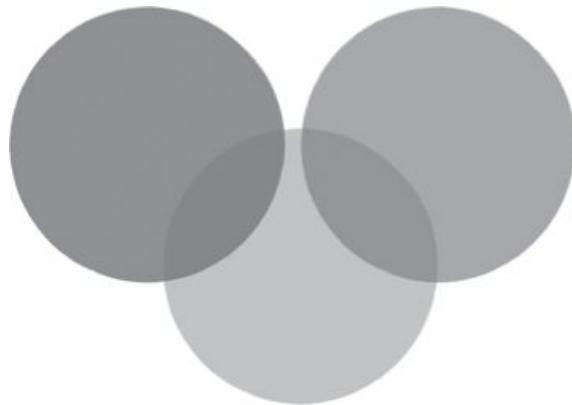
E o que você tem ao combiná-los: pessoas.

Neste capítulo, daremos uma olhada em cada um desses itens e veremos o que é possível aprender sobre como escrever um software melhor.

Círculos

A primeira parte da complexidade do software que devemos considerar está relacionada aos *círculos*, ou seja, aos componentes que criamos. O tamanho e a quantidade desses círculos determinam a complexidade.

Parte da complexidade do software é uma consequência natural de seu tamanho; quanto maior se tornar um projeto, mais círculos serão necessários, mais difícil será compreendê-los e trabalhar com eles. Essa é uma complexidade *necessária*.

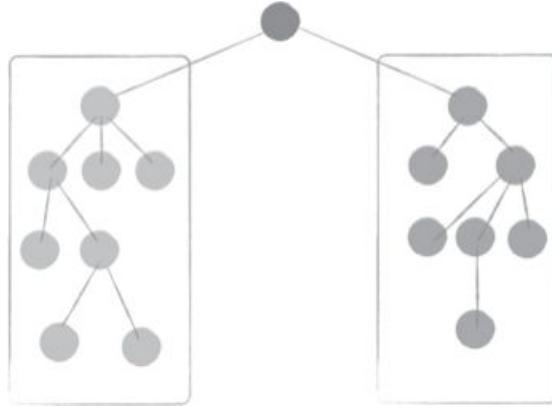
Entretanto há muita complexidade desnecessária que provoca dores de cabeça. Já perdi a conta das vezes em que abri um arquivo de header em C++ e fiquei desanimado com milhares de linhas na declaração de uma única classe. Como é que se espera que um simples mortal seja capaz de entender o que uma fera como essa faz? Certamente, essa é uma complexidade *desnecessária*.

Às vezes, esses monstros enormes são gerados automaticamente por "assistentes" (wizards) de código (exemplos dignos de nota são as ferramentas de construção de GUI). No entanto não são somente as ferramentas que devem ser culpadas. Verdadeiros fanáticos por código podem gerar essas monstruosidades sem pensar duas vezes. (Com efeito, a falta de planejamento geralmente causa esses tipos de abominação.)

Portanto devemos administrar a nossa complexidade *necessária*. E educar – ou atirar em – nossos programadores desnecessários.

É importante perceber que o tamanho em si *não é* o inimigo. Se você tiver um sistema de software que deva realizar três tarefas, então você deverá inserir código no sistema para realizar essas três tarefas. Se parte desse código for removido para reduzir a complexidade, você terá problemas diferentes. (Isso é ser *simplista* em vez de ser simples, e não é algo bom.)

Não, o tamanho por si só não é o problema. Precisamos ter código suficiente para atender aos requisitos. O problema é como estruturamos esse código e como esse tamanho é distribuído.

Suponha que você tenha começado a trabalhar em um sistema *vasto*. E você descobre que a estrutura de classe da fera é como a imagem mostrada a seguir:

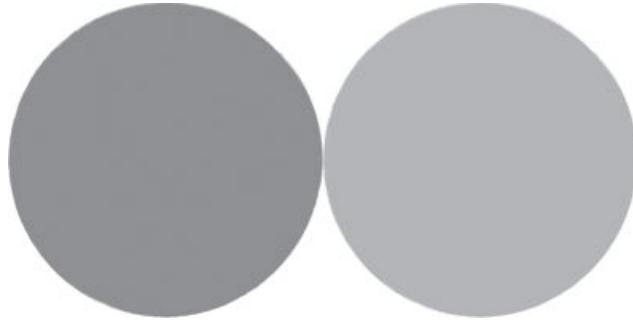
Três classes inteiras! Esse sistema é complexo ou não?

Em certo nível, ele não parece nem um pouco complicado. Há somente três partes! Como isso poderia ser difícil de entender? E o design do software tem a vantagem adicional de se parecer com o Mickey Mouse, portanto deve ser bom.

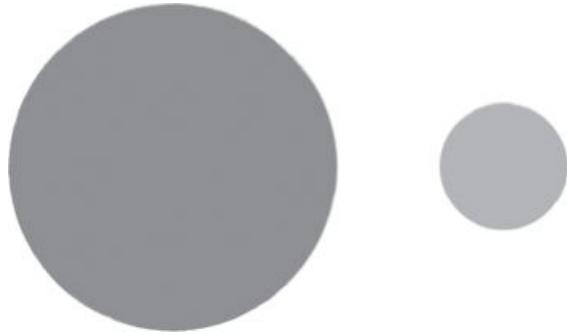
Com efeito, parece ser um design maravilhosamente simples. Você poderia descrevê-lo a alguém em segundos.

É claro, porém, que cada uma dessas partes será tão grande e densa, presumivelmente com muitas interconexões e lógica macarrônica, que provavelmente será quase impossível trabalhar com elas. Portanto é quase certo que esse seja um sistema *muito* complexo, oculto por trás de um design *simplista*.

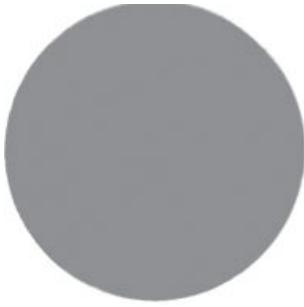
Evidentemente, uma estrutura melhor – uma que seja mais fácil de

entender e de manter – consideraria essas três seções como "módulos", subdividindo-as em outras partes: pacotes, componentes, classes ou qualquer que seja a abstração que faça sentido. Será algo mais parecido com a imagem a seguir:

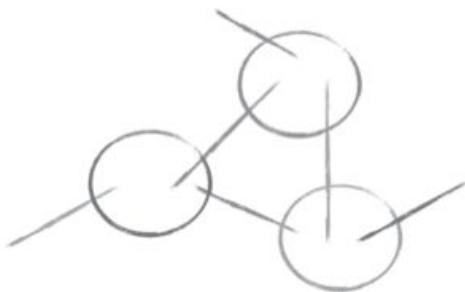
De imediato, essa imagem parece ser melhor. Ela parece ter vários componentes pequenos (portanto mais fáceis de compreender e provavelmente mais simples) conectados em um todo mais amplo. Nosso cérebro foi criado para dividir os problemas em hierarquias como essa e raciocinar sobre os problemas quando estiverem assim abstraídos.

As consequências de um design como esse são melhor compreensão e mais capacidade de modificação (você pode trabalhar em uma parte da funcionalidade do sistema ao identificar a menor parte relacionada a ela, em vez de arregaçar as mangas e mergulhar em uma única classe gigantesca). Preferimos ter classes com mais *coesão*, que façam bem várias tarefas pequenas – de preferência, somente *uma*.

É claro que o truque para fazer isso funcionar – o truque que permite que um design como esse *seja* realmente simples em vez de somente *parecer* simples – é garantir que cada um desses círculos tenha as *funções e responsabilidades* corretas. Isso significa que uma única responsabilidade deve estar em uma única parte do sistema, e não espalhada por ele.

Caso de estudo: reduzindo a complexidade dos círculos

Uma de minhas recentes reduções favoritas de complexidade de software ocorreu em uma seção de código com dois objetos bem grandes que

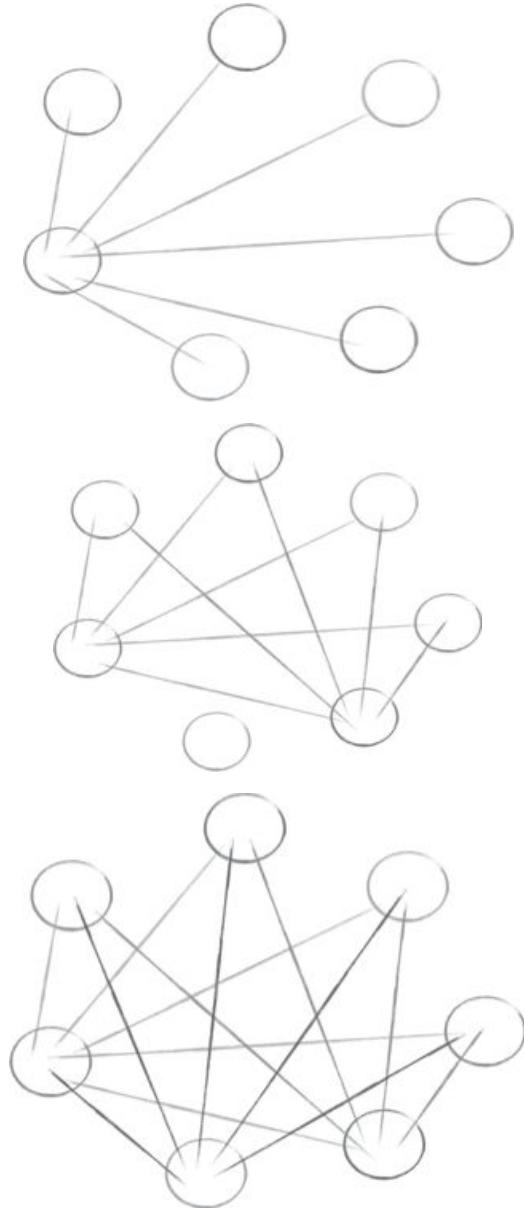
estavam tão inter-relacionados que, praticamente, eram uma mesma classe.

Comecei a fazer remoções em um dos objetos, percebendo que ele continha centenas de métodos “auxiliares” não utilizados. Removi esses métodos sem dó nem piedade; uma experiência agradável, como esvaziar um balão de gás hélio. E, como consequência, comecei a falar com uma voz aguda mais animada. Era o código se tornando mais simples.

Agora que eu podia ver o restante do objeto, estava claro que a maior parte de seus métodos simplesmente fazia um encaminhamento para o seu parceiro. Portanto removi esses métodos e fiz com que todo o código que os chamava usassem o outro objeto. Havia somente dois métodos restantes, um dos quais pertencia ao parceiro de qualquer maneira, e outro que deveria ser uma função simples, não pertencente à classe.

O resultado?

Um design de classe muito mais simples; acho que você concordará comigo.

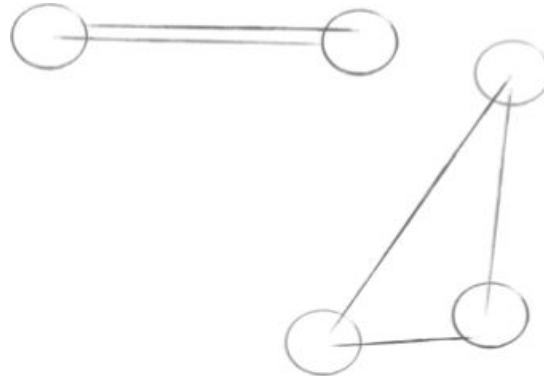
É claro que o próximo passo foi decompor o círculo restante. Mas essa é outra história. (E nem chega perto de ser tão interessante.)

Linhas

Já consideramos os círculos: os componentes e os objetos que criamos. Para parafrasear John Donne: *nenhum código é uma ilha*. A complexidade não nasce somente dos círculos, mas da maneira como são conectados.

Em geral, os designs de software são mais simples quando há menos linhas. Quanto mais conexões houver entre os círculos (isso é conhecido como mais *acoplamento* se você estiver falando devidamente como um adulto), mais rígido será o design e mais interoperações você deverá englobar (e lutar contra) à medida que trabalhar em um sistema.

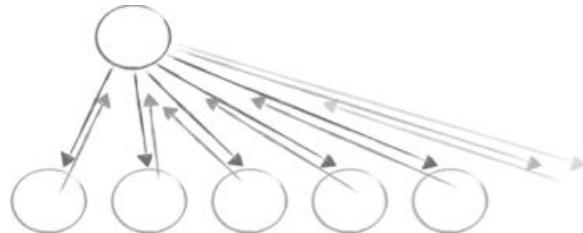
No nível mais básico, um sistema composto de vários objetos, em que nenhum deles está conectado, parecerá ser o mais simples dos sistemas. Porém, definitivamente, esse não será um único sistema. Serão vários sistemas separados.

À medida que adicionarmos conexões, criaremos verdadeiros sistemas de software. À medida que adicionarmos mais círculos e,

fundamentalmente, mais linhas entre eles, mais complexo se tornará o nosso sistema.

A estrutura de nossas interconexões de software afeta dramaticamente a facilidade com que trabalhamos com ele. Considere as estruturas a seguir, baseadas em exemplos reais com os quais já trabalhei.

Qual é a sua reação diante deles? Qual deles parece ser mais simples? Devo admitir que trabalhar com o último quase fez minha cabeça explodir.

Quando mapeamos as conexões, vemos que a complexidade em geral se

origina de ciclos em nosso gráfico. Dependências cíclicas, em geral, são os relacionamentos mais complexos a serem considerados. Quando os objetos são codependentes, sua estrutura é rígida, difícil de modificar, e, com frequência, é muito complicado trabalhar com eles. Uma mudança em um objeto geralmente exigirá uma alteração no outro. Os objetos realmente se tornam uma entidade e será mais difícil de manter.

Esses tipos de relacionamento podem ser simplificados se rompermos uma das ligações. Talvez introduzindo novas interfaces abstratas para reduzir o acoplamento entre os objetos.

Esse tipo de estrutura melhora a capacidade de composição, introduz flexibilidade e estimula a testabilidade (você pode escrever versões dos componentes para teste por trás dessas interfaces abstratas). Podemos usar interfaces bem nomeadas para tornar esses relacionamentos descritivos.

Um dos piores sistemas com que já tive de trabalhar durante muito tempo se parecia com:

Parece ser um modelo aparentemente simples: um objeto-pai representa "o sistema" e cria todos os objetos-filhos. Entretanto cada um desses objetos recebia uma referência de volta para o pai para que pudessem

acessar um ao outro. Esse design permitia que todo objeto-filho dependesse (e se tornasse altamente acoplado) de todos os irmãos, engessando todo o sistema em um formato rígido.

Michael Feathers descreveu isso para mim como o conhecido antipadrão *distributed self* (eu distribuído). Eu tinha outro nome para ele, porém não é educado o suficiente para ser impresso.

E por fim: as pessoas

Então a complexidade do software depende da estrutura de nossos círculos e de nossas linhas.

Mas é importante observar que os círculos e as linhas não se criam sozinhos. Essas estruturas não devem ser intrinsecamente culpadas. As responsáveis são as *pessoas que escrevem o código* (sim, é você, meu caro leitor). É o programador que tem o poder de introduzir uma complexidade incrível ou de reduzir um problema terrível a uma solução simples e elegante.

Com que frequência as pessoas se propõem a escrever um código horrível e complexo? Você pode achar que seus colegas de trabalho corruptos estão planejando introduzir mais estresse em sua vida com seus códigos maquiavélicos. Porém a complexidade geralmente é acidental; *raramente* será algo introduzido de propósito.

Geralmente, ela é o produto da história: os programadores estendem e estendem e estendem o sistema, sem ter tempo de refatorá-lo. Ou o "protótipo a ser jogado fora" se transforma em um sistema de produção. Quando esse código estiver sendo usado, não haverá mais chances de acabar com ele e começar novamente.

A complexidade do software é o resultado de pessoas trabalhando em situações do mundo real. A única maneira de reduzir a complexidade é nos responsabilizarmos por ela e tentar evitar que as pressões no trabalho forcem nosso código a ter estruturas com as quais seja impossível trabalhar.

Conclusão

Nesse pequeno passeio pelo território da complexidade de software, vimos que a complexidade surge dos círculos (nossos componentes de software), das linhas (as conexões entre esses componentes), mas, acima de tudo, das pessoas que criam essas monstruosidades de software.

Ah, e é claro, do padrão de projeto Singleton. Mas ninguém mais usa isso, certo?

Perguntas

1. Por que a simplicidade no código representa um design melhor? Há uma diferença entre simplicidade no *design* e na *implementação* do código?
2. Como você pode se esforçar para ter simplicidade em seu código? Como você saberá que isso foi atingido?
3. A *natureza* das conexões importa tanto quanto a quantidade delas? Quais conexões são "melhores" que as outras?
4. Se a complexidade do software tem origem em problemas sociais, como podemos abordar isso?
5. Como reconhecer a diferença entre a complexidade *necessária* e a complexidade *desnecessária*?
6. Se for verdade que muitos programadores *sabem* que os designs de seus softwares deveriam ser mais simples, como podemos incentivá-los a compor códigos mais simples?

Veja também

- *Mantenha a simplicidade* (Capítulo 16) – O outro lado da complexidade, ou seja, a *simplicidade*. Esse capítulo discute algumas ideias para a criação de designs simples.
- *O poder das pessoas* (Capítulo 34) – São as *pessoas* que criam a complexidade. Procure trabalhar com o tipo de pessoa que reduza a desordem, em vez de promovê-la.
- *Chafurdando na lama* (Capítulo 7) – Uma complexidade desnecessária

resulta em um código confuso, difícil de entender.

- *Um estudo sobre reutilização de código* (Capítulo 19) – Empregar a estratégia certa de *reutilização de código* pode ajudar a reduzir a complexidade. A estratégia errada criará um lamaçal complexo.
- *Um conto de dois sistemas* (Capítulo 13) – Um exemplo de design complexo em oposição a um design simples, mostrando as consequências de cada em deles.

> **TENTE ISTO...** Identifique maneiras pelas quais você tenha introduzido uma complexidade desnecessária em um código recente. Como isso pode ser tratado?

10.000 MACACOS
(OU ALGO POR AÍ)

CAPÍTULO 13
Um conto de dois sistemas

A arquitetura é a arte de desperdiçar espaço.

– Philip Johnson

Um sistema de software é como uma cidade – uma rede intrincada de rodovias e hotelaria, estradas secundárias e prédios. Há muitas atividades ocorrendo em uma cidade movimentada; fluxos de controle nascem a todo instante, entrelaçando suas vidas pela cidade e depois morrendo. Uma imensidão de dados é reunida, armazenada e destruída. Há uma variedade de prédios; alguns são altos e bonitos, outros são baixos e funcionais e outros ainda estão dilapidados, caindo em ruína. À medida que os dados fluem em torno deles, há congestionamentos e filas, horas de rush e obras nas estradas. A qualidade de sua cidade de software está diretamente relacionada ao nível de planejamento.

Alguns sistemas de software têm sorte: tiveram um design planejado, feito por arquitetos experientes. Estão estruturados com um senso de elegância e de equilíbrio. Estão bem mapeados e é fácil navegar por eles. Outros não têm tanta sorte assim – são uma vila de software que cresceu em torno de uma reunião acidental de códigos. A infraestrutura de transporte é inadequada e os prédios são desgastados e nada inspiradores. Se estivesse no meio de um local como esse, você estaria totalmente perdido, tentando encontrar uma rota de saída.

Em que lugar você gostaria de colocar o seu código? Que tipo de cidade de software você gostaria de construir?

Neste capítulo, narrarei a história de duas cidades de software como essas. É uma história real, e, como todas as boas histórias, essa tem uma moral no final. Dizem que a *experiência é um excelente mestre*, porém a experiência de outras pessoas é melhor ainda; se puder aprender a partir

dos erros e sucessos desses projetos, você poderá evitar muito sofrimento – e seu software também.

Esses dois sistemas são particularmente interessantes porque tiveram resultados bem diferentes, apesar de serem superficialmente muito parecidos, terem tamanhos de código, domínio do produto e nível de experiência dos engenheiros bem semelhantes.

Nessa história, os nomes foram alterados para proteger os inocentes. E os culpados.

A Metrópole Confusa

Construa, construa, prepare o caminho! Remova os obstáculos do caminho de meu povo.

– Isaías 57:14

O primeiro sistema de software em que daremos uma olhada é conhecido como a Metrópole Confusa. É um sistema do qual me recordo com carinho – não porque fosse bom nem porque fosse agradável trabalhar com ele, mas porque me ensinou uma lição importante sobre desenvolvimento de softwares quando o conheci.

Juntei-me ao projeto Metrópole quando ele já estava “maduro”. Era uma base de código complexa, e conhecê-la exigia um tempo incrivelmente longo. No nível micro, as linhas de código eram confusas e inconsistentes. No nível macro, o design era confuso e inconsistente.

Ninguém da equipe realmente conhecia o funcionamento de todo o código. Esse havia crescido “organicamente” durante anos (que é uma maneira bem educada de dizer que ninguém havia feito um design de arquitetura digno de nota e que várias porções de código haviam sido inseridas com o passar do tempo, sem que ninguém tivesse pensado muito sobre o assunto). Ninguém havia jamais parado para impor uma estrutura razoável ao código. Ele havia crescido por meio de acumulação; era um exemplo clássico de um sistema que não havia recebido absolutamente nenhum design de arquitetura. Entretanto uma base de código não pode ter *nenhuma* arquitetura. Simplesmente era uma

arquitetura bem pobre.

O estado do projeto Metrópole era compreensível (porém não perdoável) se olhássemos a história da empresa que o havia desenvolvido: uma start-up altamente pressionada no sentido de disponibilizar muitas versões novas rapidamente. Atrasos não eram tolerados – eles significariam a ruína financeira. Os engenheiros de software eram orientados a disponibilizar códigos para as novas versões tão rapidamente quanto fosse humanamente possível (ou mais rápido ainda). E, assim, o código havia sido reunido em uma série de arrancadas malucas.

> **PONTO-CHAVE** Uma estrutura de empresa pobre e processos de desenvolvimento que não sejam saudáveis se refletirão em uma arquitetura de software pobre.

A falta de planejamento do projeto Metrópole trouxe diversas consequências que veremos aqui. Essas ramificações foram graves e avançaram muito além do que você poderia esperar ingenuamente de um design ruim.

Incompreensibilidade

A arquitetura do Metrópole, com a falta de uma estrutura imposta, havia resultado em um sistema de software incrivelmente difícil de compreender e praticamente impossível de modificar. Novos recrutas que entravam no projeto (como eu) ficavam abismados com a complexidade e eram incapazes de entender o que estava acontecendo.

O design ruim incentivava outros designs ruins a serem acrescentados – com efeito, você era literalmente forçado a fazer isso –, pois não havia nenhum meio de estender o design de maneira sensata. O caminho mais fácil para realizar a tarefa em mãos era sempre adotado; não havia nenhuma maneira óbvia de corrigir os problemas estruturais e, portanto, as novas funcionalidades eram jogadas onde quer que causassem menos dores de cabeça.

> **PONTO-CHAVE** Mantenha a qualidade do design de um software. Um design ruim resulta em mais designs ruins.

Falta de coesão

Os componentes do sistema não eram nada coesos. Nos pontos em que cada componente deveria ter uma função única e bem definida, havia uma reunião de funcionalidades que não estavam necessariamente relacionadas. Isso dificultava determinar o motivo pelo qual um componente existia, antes de tudo, e era difícil descobrir em que local uma determinada funcionalidade havia sido implementada no sistema.

Naturalmente, isso tornava a correção de bugs um pesadelo. A qualidade e a confiabilidade do software eram seriamente afetadas.

Tanto as funcionalidades quanto os dados estavam no lugar errado no sistema. Muitos dos que eram considerados "serviços essenciais" não estavam implementados no núcleo do sistema, mas eram simulados por módulos mais externos (com muito custo e dificuldade).

Uma arqueologia de software mostrou o motivo: houve conflitos de personalidade na equipe original de modo que alguns programadores principais começaram a criar seus pequenos impérios de software. Eles colocavam as funcionalidades que achavam ser interessantes em seus módulos, mesmo que elas não pertencessem a esse local. Para lidar com isso, esses programadores criavam sistemas de comunicação mais esdrúxulos ainda, de modo a devolver o controle para o local correto.

PONTO-CHAVE A saúde dos relacionamentos profissionais em sua equipe de desenvolvimento terá consequências diretas no design do software. Relacionamentos não saudáveis e egos inflados resultam em um software não saudável.

Coesão e acoplamento

Duas qualidades fundamentais do design de software são *coesão* e *acoplamento*. Esse não é um conceito ultramoderno de "OO"; os desenvolvedores falam disso há anos (desde o surgimento do design estruturado no início da década de 70). Temos como meta efetuar o design de sistemas com componentes que tenham:

Forte coesão

A coesão é uma medida de como funcionalidades relacionadas estão reunidas e de como as partes *dentro* de um módulo funcionam

adequadamente como um todo. A coesão é a cola que une um módulo.

Os módulos com coesão fraca são um sinal de uma decomposição ruim. Cada módulo deve ter uma função claramente definida, e não uma reunião de funcionalidades não relacionadas.

Baixo acoplamento

O acoplamento é uma medida da interdependência *entre* os módulos; a quantidade de conexões de e para eles. Nos designs mais simples, os módulos têm baixo acoplamento, portanto dependem menos uns dos outros. Obviamente, os módulos não podem ser totalmente desacoplados, ou não funcionariam em conjunto!

Os módulos se interconectam de diversas maneiras; algumas diretas e outras indiretas. Um módulo pode chamar funções de outros módulos ou pode ser chamado por outros módulos. Pode usar web services ou recursos publicados por outro módulo. Pode usar tipos de dados de outro módulo ou compartilhar alguns dados (talvez variáveis ou arquivos).

Um bom design de software limita as linhas de comunicação àquelas que sejam absolutamente necessárias. Essas linhas de comunicação constituem parte do que determina a arquitetura.

Acoplamento desnecessário

O Metrópole não tinha uma divisão clara em camadas. As dependências entre os módulos não eram unidirecionais; com frequência, o acoplamento era bidirecional. O componente A acessava a parte interna do componente B para fazer o seu trabalho em uma tarefa. Em outro local, o componente B tinha chamadas fixas no código para o componente A. Não havia uma camada inferior ou um núcleo central para o sistema. Era um bloco monolítico de software.

Isso significava que as partes individuais do sistema eram tão fortemente acopladas que não era possível criar um esqueleto do sistema sem que todos os componentes fossem criados. Qualquer alteração em um único componente se propagava, exigindo mudanças em vários componentes dependentes. Os componentes de código não faziam sentido

isoladamente.

Isso tornava impossível os testes de baixo nível. Não só era impossível escrever testes de unidade no nível de código como também os testes de integração no nível de componentes não podiam ser criados, pois todo componente dependia de quase todos os demais componentes. É claro que testar nunca havia sido particularmente uma alta prioridade na empresa (não tínhamos nem perto do tempo suficiente para isso), portanto isso "não era um problema". Nem é preciso dizer que o software não era muito confiável.

> **PONTO-CHAVE** Um bom design leva em consideração os sistemas de conexão e a quantidade (e a natureza) das conexões entre os componentes. As partes individuais de um sistema devem ser capazes de ser independentes. Um alto acoplamento resulta em um código impossível de ser testado.

Problemas com o código

Os problemas com um design de alto nível ruim haviam se esgueirado para baixo em direção ao nível do código. Problemas geram problemas. Como não havia um design comum e nenhum "estilo" geral de projeto, ninguém se preocupava com padrões de codificação, bibliotecas ou o emprego de idioms comuns. Não havia nenhuma convenção de nomenclatura para componentes, classes ou arquivos. Não havia nem mesmo um sistema comum de build; fitas adesivas, shell scripts e colas com Perl se entrelaçavam com makefiles e arquivos de projeto do Visual Studio. Compilar esse monstro era considerado um rito de passagem!

Um dos problemas mais sutis, porém mais sérios, do Metrópole era a duplicação. Sem um design claro e um local evidente para inserir as funcionalidades, as rodas haviam sido reinventadas em toda a base de código. Tarefas simples como algoritmos comuns e estruturas de dados eram repetidas em diversos módulos; cada implementação tinha o seu próprio conjunto de bugs obscuros e traços comportamentais peculiares. Responsabilidades de mais larga escala como comunicação externa e caching de dados também eram implementadas diversas vezes.

Uma arqueologia de software adicional mostrou o motivo: o Metrópole

havia começado como uma série de protótipos separados que foram reunidos, quando deveriam ter sido jogados fora. O Metrópole era, na realidade, uma conurbação acidental. Quando foram reunidos, os componentes de código jamais se encaixaram adequadamente. Com o tempo, as costuras mal feitas começaram a se desfazer e os componentes começaram a deslizar uns contra os outros e a provocar atrito na base de código, em vez de trabalharem em harmonia.

> **PONTO-CHAVE** Uma arquitetura descuidada e frágil resulta em componentes de código individuais mal escritos e que não se encaixam bem. Também resulta em duplicação de código e de esforços.

Problemas além do código

Os problemas do Metrópole se espalharam para além da base de código e provocaram confusão em outros locais na empresa. Havia problemas na equipe de desenvolvimento, porém a arquitetura ruim também afetava as pessoas que davam suporte e que usavam o produto.

A equipe de desenvolvimento

Novos recrutas que entravam no projeto (como eu) ficavam abismados com a complexidade e eram incapazes de entender o que estava acontecendo. Isso explica parcialmente por que bem poucos recrutas permaneciam na empresa por muito tempo – a rotatividade da equipe era muito alta.

Aqueles que permaneciam tinham de trabalhar arduamente – os níveis de estresse no projeto eram altos. Planejar novas funcionalidades incutia um medo terrível.

Ciclo lento de desenvolvimento

Manter o Metrópole era uma tarefa assustadora, de modo que até mesmo alterações simples ou *pequenas* correções de bugs exigiam uma quantidade de tempo imprevisível. Administrar o ciclo de desenvolvimento do software era difícil, planejar o cronograma era complicado, e o ciclo de disponibilização de versões era instável e lento. Os clientes ficavam à espera de funcionalidades importantes e a gerência ficava cada vez mais frustrada com a incapacidade da equipe

de desenvolvimento de atender aos requisitos de negócio.

Engenheiros de suporte

Os engenheiros que trabalhavam com o suporte do produto tinham muita dificuldade em dar suporte a um produto frágil, ao mesmo tempo que tentavam descobrir as diferenças comportamentais intrincadas entre versões de software com diferenças relativamente pequenas.

Suporte de terceiros

Um protocolo de controle externo havia sido desenvolvido, permitindo que outros dispositivos controlassem o Metrópole remotamente. Como essa era somente uma camada superficial fina sobre as entranhas do software, ela refletia a arquitetura do Metrópole, o que significa que era esdrúxula, difícil de entender, suscetível a falhas aleatórias e impossível de usar. A vida dos engenheiros de produtos de terceiros também se tornou miserável por causa da estrutura pobre do Metrópole.

Políticas internas da empresa

Os problemas de desenvolvimento resultaram em atrito entre diferentes grupos na empresa. A equipe de desenvolvimento tinha relações ruins com o pessoal de marketing e de vendas, e o departamento de produção vivia permanentemente estressado sempre que uma versão nova surgia no horizonte. Os gerentes se desesperavam.

PONTO-CHAVE As consequências de uma arquitetura ruim não ficam restritas ao código. Elas se espalham e afetam pessoas, equipes, processos e cronogramas.

Um cartão-postal de Metrópole

O design do Metrópole era praticamente irrecuperável – acredite, nós tentamos corrigi-lo. O nível de esforço exigido para retrabalhar, refatorar e corrigir os problemas da estrutura do código havia se tornado proibitivo. Uma reescrita não era uma opção barata.

O "design" do Metrópole teve como consequência uma situação diabólica que piorava inexoravelmente. Era tão difícil acrescentar funcionalidades novas que as pessoas estavam simplesmente aplicando mais gambiarras, esparadrapos e remendos calculados. Ninguém gostava de trabalhar com o código e o projeto estava caindo em uma espiral crescente. A falta de design havia resultado em um código ruim, que resultou em baixo moral da equipe e em ciclos cada vez mais longos de desenvolvimento. Posteriormente, houve sérios problemas financeiros na empresa.

Uma arquitetura ruim tem um efeito profundo e gera repercussões sérias. A falta de visão e de design no projeto Metrópole Confusa resultou em:

- baixa qualidade do produto, com poucas versões disponibilizadas;
- um sistema inflexível que não podia acomodar alterações nem a adição de novas funcionalidades;
- problemas de código em todos os lugares;
- problemas com a equipe (estresse, moral, rotatividade);
- muita confusão na política interna da empresa;
- fracasso para a empresa;
- muitas dores de cabeça sérias e trabalhos no código que avançavam noite adentro.

Cidade do Design

A forma sempre segue a função.

– Louis Henry Sullivan

O projeto de software Cidade do Design, superficialmente, era muito semelhante à Metrópole Confusa. Era um produto semelhante implementado com as mesmas tecnologias. Entretanto esse projeto havia sido construído de maneira bem distinta e sua estrutura interna era muito diferente.

O projeto Cidade do Design havia sido criado do zero por um grupo pequeno de programadores. Assim como no Metrópole, a estrutura da equipe era linear. Felizmente, não havia nenhuma rivalidade entre as

pessoas nem disputas por posições de poder na equipe. Desde o princípio, havia uma visão clara do produto inicial e um conjunto de requisitos para ele.

Uma direção inicial para o design havia sido definida (não era um design prévio grande, mas *apenas suficiente* para trabalhar). As principais áreas funcionais haviam sido demarcadas e algumas preocupações centrais com a arquitetura, como os modelos de threading, haviam sido esquematizadas. As áreas funcionais mais importantes receberam atenção inicial no design.

As decisões sobre algumas das preocupações mais básicas com manutenção foram tomadas bem cedo para garantir que o código crescesse de forma fácil e coesa: a estrutura de arquivos de mais alto nível, a nomenclatura dos itens, um estilo "doméstico" de apresentação com idioms comuns de codificação, a escolha do framework de testes de unidade e a infraestrutura de suporte. Esses *detalhes finos* eram muito importantes e influenciaram muitas das decisões futuras de design.

O design e a implementação do código foram feitos aos pares ou eram cuidadosamente revisados para garantir que o trabalho estivesse correto. O design e o código se desenvolveram e amadureceram com o tempo e, à medida que a história da Cidade do Design se desenrolava, houve consequências.

Localizando as funcionalidades

Com uma visão geral clara da estrutura do sistema definida desde o princípio, novas unidades de funcionalidades eram consistentemente adicionadas nas áreas funcionais corretas da base de código. Jamais havia um questionamento sobre o local a que um código pertencia. Também era fácil encontrar a implementação de funcionalidades existentes para estendê-la ou para corrigir problemas.

Às vezes, porém, inserir um código novo no lugar *certo* era mais difícil do que simplesmente colocá-lo em um local mais conveniente, porém menos adequado. Portanto a existência de um plano de arquitetura às vezes fazia com que os desenvolvedores tivessem de trabalhar mais arduamente. A

recompensa por esse esforço extra era uma vida *muito* mais simples no futuro, para manter ou para estender o sistema – havia pouco código ruim em que tropeçar.

Uma arquitetura ajuda a localizar as funcionalidades: ajuda a adicioná-las, modificá-las ou corrigi-las. Fornece um template em que o trabalho pode ser encaixado e um mapa para navegar pelo sistema.

Consistência

O sistema todo era consistente. Toda decisão em qualquer nível era tomada no contexto do design como um todo. Os desenvolvedores fizeram isso intencionalmente desde o princípio para que todo o código produzido estivesse totalmente de acordo com o design e com todos os demais códigos escritos.

Ao longo da história do projeto, apesar das diversas alterações por todo o escopo da base de código – de linhas individuais até a estrutura do sistema – tudo seguia o template original do design.

> **PONTO-CHAVE** Um design claro de arquitetura resulta em um sistema consistente. Todas as decisões devem ser tomadas no contexto do design da arquitetura.

O bom gosto e a elegância do design de mais alto nível naturalmente se propagaram para os níveis inferiores. Mesmo nos níveis mais baixos, o código era uniforme e organizado. Um design de software definido claramente garantiu que não houvesse nenhuma duplicação, que padrões de projeto familiares fossem usados em todos os lugares, que idioms de interface familiares fossem adotados e que não houvesse durações incomuns da vida de objetos nem problemas estranhos de gerenciamento de recursos. As linhas de código eram escritas no contexto do planejamento da cidade.

> **PONTO-CHAVE** Uma arquitetura clara ajuda a reduzir a duplicação de funcionalidades.

Expandindo a arquitetura

Algumas áreas funcionais completas apareceram no design do "quadro geral" – gerenciamento de armazenamento de dados e um recurso de controle externo, por exemplo. No projeto Metrópole, esses foram problemas sérios e incrivelmente difíceis de implementar. Porém, na Cidade do Design, o funcionamento era diferente.

O design do sistema, assim como o código, era considerado maleável e passível de refatoração. Um dos princípios fundamentais da equipe de desenvolvimento era permanecer ágil – nada estava gravado a ferro e fogo – e, portanto, a arquitetura deveria ser alterada quando fosse necessário. Isso nos estimulava a manter nossos designs simples e fáceis de ser alterados. Consequentemente, o código podia crescer rapidamente e uma boa estrutura interna podia ser mantida. Acomodar novos blocos funcionais não era um problema.

PONTO-CHAVE A arquitetura do software não está gravada a ferro e fogo. Mude-a se for necessário. Para pode ser alterável, a arquitetura deve permanecer simples. Resista a mudanças que comprometam a simplicidade.

Adiando as decisões de design

Um princípio da XP que melhorou a qualidade da Cidade do Design foi o YAGNI[1] (não faça nada se *não for precisar*). Esse princípio nos incentivou a fazer inicialmente apenas o design daquilo que fosse importante e adiar todas as decisões restantes – para quando tivéssemos uma visão mais clara dos verdadeiros requisitos e soubéssemos como encaixá-los da melhor maneira possível no sistema. Essa é uma abordagem de design extremamente eficiente e muito libertadora:

- Uma das piores tarefas que você pode fazer é criar o design de algo que você ainda não conhece. O YAGNI força você a esperar até que o problema seja realmente conhecido e você saiba de que modo o design deverá acomodá-lo. Ele elimina os palpites e garante que o design estará correto.
- É perigoso adicionar tudo de que você *poderá* precisar (incluindo a pia da cozinha) em um design de software ao criá-lo. A maioria de seu trabalho de design será um desperdício de esforço, ou seja, uma

bagagem extra à qual você deverá dar suporte durante toda a vida mutante do software. O custo será mais alto no início e continuará aumentando no decorrer da vida do projeto.

> **PONTO-CHAVE** Adie as decisões de design até que seja *necessário* tomá-las. Não tome decisões de arquitetura enquanto os requisitos ainda não forem conhecidos. Não adivinhe.

Mantendo a qualidade

Desde o princípio, o projeto Cidade do Design definiu diversos processos de controle de qualidade:

- programação aos pares;
- revisões de código/design para tudo que não fosse feito aos pares;
- testes de unidade para todas as partes do código.

Isso garantia que uma alteração incorreta, que não se encaixasse bem no sistema, jamais fosse aplicada. Tudo que não se entrosasse com o design do software era rejeitado. Isso pode parecer draconiano, porém eram processos com os quais os desenvolvedores haviam se comprometido.

Esse comprometimento enfatiza uma atitude importante: os desenvolvedores acreditavam no design e o consideravam importante o suficiente para protegê-lo. Eles se consideravam proprietários do design e assumiam uma responsabilidade pessoal por ele.

> **PONTO-CHAVE** A qualidade do design deve ser mantida. Isso pode acontecer somente quando os desenvolvedores assumem seriamente a responsabilidade por ele.

Administrando a dívida técnica

Apesar dessas medidas de controle de qualidade, o desenvolvimento da Cidade do Design foi bem pragmático. À medida que os prazos se esgotavam, vários atalhos foram tomados para permitir que o projeto fosse disponibilizado a tempo. Permitiu-se que pequenos *pecados* de código ou pequenas imperfeições no design fossem inseridos na base de código, seja para que uma funcionalidade funcionasse rapidamente ou

para evitar mudanças de alto risco próximas a uma disponibilização de versão.

Entretanto, de modo diferente do projeto Metrópole Confusa, essas deficiências eram marcadas como *débitos técnicos* e uma revisão futura era agendada. Essas imperfeições ficavam claramente destacadas, e os desenvolvedores não ficavam satisfeitos até que elas fossem tratadas. Novamente, vemos os desenvolvedores assumindo a responsabilidade pela qualidade do design.

Débito técnico

Débito técnico é um termo cunhado por Ward Cunningham, amplamente usado no mercado de software hoje em dia. A metáfora está calcada no mundo financeiro: tomar uma decisão para ajudar a disponibilizar rapidamente o software é como fazer um empréstimo. Ele pode permitir que você faça algo *agora* que, de outra maneira, não poderia ter sido feito.

Porém esse empréstimo não pode ser ignorado – ele sempre deverá ser pago. Quanto mais tempo você demorar a efetuar o pagamento, mais alto será o custo. Se não fizer os pagamentos no prazo, você acabará pagando juros pelo empréstimo e o seu poder de compra diminuirá.

No mundo do software, isso significa retornar ao seu código para atualizá-lo; do contrário, seu progresso se tornará mais lento à medida que o código afundar em dívidas acumuladas. Isso é importante: no longo prazo, uma qualidade mais baixa do código significa prazos mais longos de desenvolvimento, porém um empréstimo responsável de curto prazo *pode* agilizar a situação.

A dívida técnica pode ser uma refatoração adiada, ajustes no design que reflitam o que você descobriu, espera para atualizar bibliotecas ou conjuntos de ferramentas até a próxima disponibilização de versão importante ou a racionalização da estrutura de logging/debugging.

É uma metáfora rica que pode ser usada erroneamente: uma dívida técnica *não* significa somente fazer algo de forma ruim. Às vezes, escrever um código ruim é justificado como sendo "pragmático"; há uma diferença entre uma opção pragmática e uma opção descuidada.

Administrar a dívida técnica com consciência é uma arma eficaz em seu

arsenal de desenvolvimento. Não deixe que a dívida se acumule e mantenha-a visível. Como um verdadeiro empréstimo, pague a dívida o mais cedo possível para evitar um sofrimento com cobranças e juros excessivos.

Os testes moldam o design

Uma de nossas principais decisões foi assumir que testes de unidade deveriam ser feitos no código, e testes de integração e de aceitação deveriam ser feitos no sistema. Os testes de unidade apresentam muitas vantagens, uma das quais é a capacidade de alterar seções do software sem nos preocuparmos com a destruição de tudo o mais no processo.

Algumas áreas da estrutura interna da Cidade do Design passaram por um retrabalho bem radical, ao mesmo tempo que os testes de unidade nos deram a certeza de que o restante do sistema continuava funcionando. Por exemplo, o modelo de thread e a interface para interconexão do pipeline de dados foram radicalmente alterados. Foi uma mudança séria de design relativamente tarde no desenvolvimento desse subsistema, porém o restante do código que fazia interface com esse pipeline continuou a funcionar perfeitamente. Os testes nos deram a capacidade de alterar o design.

Esse tipo de alteração "grande" de design se tornou menos frequente à medida que a Cidade do Design amadurecia. Após alguns retrabalhos no design, tudo se acalmou e posteriormente houve somente algumas alterações menores. O sistema foi desenvolvido rapidamente, de maneira iterativa, com cada passo aperfeiçoando o design até que ele atingisse um platô relativamente estável.

PONTO-CHAVE Ter um bom conjunto de testes automatizados para o seu sistema permite fazer alterações fundamentais na arquitetura com um mínimo de risco. Você terá espaço para trabalhar.

Outra grande vantagem dos testes de unidade foi o fato de eles terem notadamente moldado o design do código; eles praticamente forçaram a existência de uma boa estrutura. Cada pequeno componente de código foi criado como uma entidade bem definida, que podia ser independente – ele devia poder ser construído em um teste de unidade sem que se

exigisse que o restante do sistema fosse construído em torno dele. Escrever testes de unidade garantiu que cada módulo de código fosse internamente coeso e tivesse baixo acoplamento em relação ao restante do sistema. Os testes de unidade forçaram um planejamento cuidadoso da interface de cada unidade e garantiram que sua API fosse significativa e internamente consistente.

> **PONTO-CHAVE** Fazer testes de unidade em seu código resulta em designs melhores de software, portanto crie o design visando à testabilidade.

Prazo para o design

Um dos fatores que contribuíram com o sucesso da Cidade do Design foi o prazo alocado para o desenvolvimento, que não foi nem muito longo nem muito curto (como diria Cachinhos Dourados, foi "simplesmente adequado").

Um projeto precisa de um ambiente favorável para florescer.

Se houver tempo demais, com frequência, os programadores irão querer criar seu *magnum opus* (o tipo de projeto que sempre estará *quase* pronto, porém jamais se materializará). Um pouco de pressão faz muito bem, e um senso de urgência colabora para que tudo fique pronto. No entanto, se houver pouco tempo, é simplesmente impossível criar qualquer design que valha a pena, e você terá uma solução apressada e pela metade – exatamente como no caso do projeto Metrópole.

> **PONTO-CHAVE** Um bom planejamento de projeto resulta em designs superiores. Aloque tempo suficiente para criar uma obra-prima da arquitetura – ela não surge instantaneamente.

Trabalhando com o design

Embora fosse grande, a base de código era coerente e fácil de compreender. Novos programadores podiam entendê-la e trabalhar com ela de forma relativamente fácil. Não havia nenhuma interconexão complexa e desnecessária para entender nem códigos legados estranhos com que trabalhar.

Como o código havia gerado relativamente poucos problemas e continuava sendo agradável trabalhar com ele, houve uma rotatividade muito, muito baixa dos membros da equipe. Em parte, isso se deve ao fato de os desenvolvedores assumirem que são donos do design e quererem melhorá-lo continuamente.

Foi interessante observar como a dinâmica da equipe de desenvolvimento seguia a arquitetura. Os princípios do projeto Cidade do Design obrigavam que ninguém fosse *dono* de nenhuma área do design e que qualquer desenvolvedor pudesse trabalhar em qualquer ponto do sistema. Esperava-se que todos escrevessem códigos de alta qualidade. Enquanto o Metrópole era uma vasta confusão criada por muitos programadores sem coordenação brigando entre si, a Cidade do Design era limpa e coesa, com componentes de software que cooperavam bem, criados por colegas que tinham um bom relacionamento. Em vários aspectos, a lei de Conway funcionava de forma inversa e a equipe se entendia bem, assim como o software.[2]

PONTO-CHAVE A organização de uma equipe tem um efeito inevitável no código produzido. Com o tempo, a arquitetura também afeta o modo como uma equipe trabalha em conjunto. Quando as equipes se separam, o código interage de forma desajeitada. Quando as pessoas trabalham juntas, a arquitetura está bem integrada.

E daí?

Essa história simples sobre dois sistemas de software certamente não é um tratado exaustivo sobre arquitetura de software, porém mostramos como ela afeta profundamente um projeto. Uma arquitetura influencia quase tudo que entrar em contato com ela, determinando a saúde da base de código e também das áreas ao redor. Assim como uma cidade em crescimento pode trazer prosperidade e fama para a sua área local, uma boa arquitetura de software ajudará um projeto a florescer e trará sucesso àqueles que dependam dele.

Uma boa arquitetura é produto de diversos fatores, incluindo (porém não limitados a):

- Fazer um design prévio intencional antes de trabalhar com o código. Muitos projetos falham nesse sentido antes de sequer começarem. Há uma tensão nesse caso; não devemos fazer menos design que o necessário, porém, da mesma maneira, não devemos exagerar.
- A qualidade e a experiência dos designers. (Ter cometido alguns erros antes pode ajudar a colocar você na direção certa da próxima vez! O projeto Metrópole certamente me ensinou uma ou duas lições.)
- Manter o design claramente visível à medida que o desenvolvimento progride.
- Uma equipe que receba e assuma a responsabilidade pelo design do software em geral.
- Jamais ter medo de alterar o design; nada está gravado a ferro e fogo.
- Ter as pessoas certas na equipe, incluindo designers, programadores e gerentes. Garanta que a equipe de desenvolvimento tenha o tamanho adequado. Certifique-se de que eles tenham relacionamentos profissionais saudáveis, pois esses relacionamentos inevitavelmente se refletirão na estrutura do código.
- Tomar as decisões de design nos momentos adequados, quando você conhecer todas as informações para isso. Adie as decisões de design que ainda não possam ser tomadas.
- Um bom gerenciamento de projeto, com os prazos adequados.

Perguntas

1. Qual é a melhor arquitetura de sistema que você já viu?
   - Como você percebeu que ela era boa?
   - Quais foram as consequências dessa arquitetura, tanto dentro quanto fora da base de código?
   - O que a levou a ter um design tão bom?
   - O que você aprendeu com ela?
2. Qual é a pior arquitetura de sistema que você já viu?
   - Como você percebeu que ela era ruim?
   - Quais foram as consequências dessa arquitetura, tanto dentro

quanto fora da base de código?

- Como ela chegou a esse estado?
- O que você aprendeu com ela?
- Como você resolveria os seus problemas?

3. Entre essas duas arquiteturas, em que ponto se encontra o seu projeto atual? Em quais de suas experiências anteriores você pode se basear para melhorar o código ou os processos usados para implementá-lo?

Veja também

- *Lidando com a complexidade* (Capítulo 12) – Como lidar com designs complexos (e evitá-los).
- *Mantenha a simplicidade* (Capítulo 16) – O formato de um código simples.
- *O fantasma de um código do passado* (Capítulo 5) – Como aprender com o código existente. Independentemente do quanto um sistema seja bom, você pode aprender com ele e se aperfeiçoar com base nele em seu próximo design.
- *É hora de testar* (Capítulo 11) – Os testes de unidade ajudaram a Cidade do Design a se desenvolver de maneira bem fatorada e confiável.

TENTE ISTO... Considere o modo como você descreveria o seu projeto atual a alguém de fora. De que você sente orgulho e o que pode ser melhorado? Como a sua equipe pode comemorar o que estiver sendo feito de bom? Determine o que você pode fazer agora para reforçar os pontos fracos.

10.000 MACACOS

(OU ALGO POR AÍ)

O SEGUNDO EFEITO DO SISTEMA

BJARNE STROUSTRUP

"SEMPRE DESEJEI QUE MEU COMPUTADOR FOSSE TÃO FÁCIL DE USAR QUANTO O MEU TELEFONE."

"MEU DESEJO FOI ATENDIDO, POIS NÃO CONSIGO MAIS ENTENDER O USO DO MEU TELEFONE."

---

1 N.T.: YAGNI quer dizer *You Aren't Going to Need It*, ou seja, “Você não irá precisar disso”.

2 A lei de Conway afirma que a estrutura do código segue a estrutura da equipe. Falando de modo simples, ela afirma que, *se você tiver quatro grupos trabalhando em um compilador, você terá um compilador de quatro passos*.

PARTE II

A prática leva à perfeição

Vamos agora nos afastar do código e apreciar uma visão mais geral. Inspecionaremos as *práticas* importantes que fazem parte de uma boa programação.

Estes capítulos discutem técnicas, práticas e abordagens importantes para escrever um bom código. Apresentarei regras para um comprometimento com o processo de desenvolvimento de software, maneiras de abordar a tarefa de codificação e técnicas sólidas que ajudarão você a cooperar com outros membros da equipe de desenvolvimento.

CAPÍTULO 14

Desenvolvimento de software é...

E assim, nossa vida, livre do convívio social, encontra idiomas nas árvores, nos livros, nos ribeirões que correm, sermões em pedras e o bem em tudo.

– William Shakespeare
Como gostais

O fato de que não poderei contar com meu intelecto aguçado para sempre é triste. Em algum momento no futuro, meu juízo vai se desvanecer e não serei mais o gênio astuto, erudito e humilde que sou agora. Sendo assim, preciso de um plano de aposentadoria – uma maneira de ganhar meus milhões para que eu possa viver no luxo quando envelhecer.

Meu plano original para dominar o mundo parecia tão simples que não poderia dar errado: *leite efervescente!* Entretanto, antes de ter tido a oportunidade de descobrir os mínimos detalhes da receita, recebi notícias devastadoras: o leite efervescente já havia sido inventado. Desapontado e com os direitos de patente escorrendo pelos meus dedos, voltei para a prancheta para pensar em um novo plano de aposentadoria. E, dessa vez, cheguei a um bom plano.

Esse gênio aqui relembrou as comidas clássicas de sua infância: creme e macarrão de letrinhas. Tenho certeza de que você pode ver a direção que estou seguindo: *creme de letrinhas*. Meus experimentos iniciais se provaram promissores. E quase palatáveis... é um pouco parecido com arroz-doce, porém é feito de trigo. Admito que é um sabor com que devemos nos acostumar, mas pode ser que conquiste o público.

Esse negócio de software (comida)

Um software moderno demais é como o meu creme de letrinhas: é *algo errado* escrito da *maneira errada*.

Para preparar o creme de letrinhas da maneira "certa", você deveria preparar primeiro o macarrão manualmente e fazer o creme da mesma maneira. A forma errada e trapaceira seria comprar um macarrão pronto, enxaguá-lo para tirar o molho e então jogar um creme instantâneo por cima.

Uma das opções é uma receita – um método para uma criação digna de confiança. A outra, no melhor caso, é uma maneira adequada de criar um protótipo, porém não é uma técnica de fabricação de larga escala.

Como desenvolvedores de software conscienciosos, todos nós devemos aspirar a *escrever o código certo* da *maneira certa*. Uma das características fundamentais de programadores realmente excelentes é importar-se com o software que escrevem e como o escrevem. Precisamos ter mais código artesanal feito à mão, e não um código macarrônico sem sentido.

Neste capítulo, daremos uma espiada na panela para investigar a natureza do software que criamos e verificar como podemos evitar a escrita de um código macarrônico alfanumérico. Apresentarei uma série de perguntas no caminho para aplicar as lições que aprendermos. A primeira pergunta é: *você quer se aperfeiçoar como programador? Você realmente quer escrever o código certo da maneira certa?*

Se sua resposta for "não", desista e pare de ler agora.

Então o que *é* desenvolvimento de software? Com certeza, ele é complexo, com vários aspectos que se entrelaçam. Embora este capítulo não possa ser um tratado intelectual completo sobre o desenvolvimento de software, podemos investigar algumas de suas nuances: o fato de que esse desenvolvimento é, em parte, ciência, em parte é arte, jogo, esporte, obrigação etc.

Desenvolvimento de software é... uma arte

Um grande programador, em parte, deve ser um grande artista. Mas a programação é *realmente* uma arte? Esse é um tema de debate que se mantém há muito tempo nos círculos de desenvolvimento de software. Algumas pessoas acham que a programação é uma disciplina de

engenharia; outros acham que é uma forma de arte, e outros, que fica em algum ponto no meio, considerando-a um trabalho artesanal[1] (afinal de contas, chamei meu primeiro livro de *Code Craft*).

Knuth provavelmente é o mais famoso proponente do software como uma arte, tendo nomeado sua famosa série de livros de *The Art of Computer Programming* (A arte da programação de computadores). Ele afirmou que *alguns programas são elegantes, outros são requintados, outros são brilhantes*. *Meu argumento é que é possível escrever programas grandiosos, nobres, verdadeiramente magníficos*. É de arrepiar.

Há mais no código além de bits e bytes; mais do que parênteses e chaves. Há estrutura e elegância. Há estabilidade e equilíbrio. Há um senso de bom gosto e de estética.

> **PONTO-CHAVE** Um programador deve ter bom gosto e senso de estética para escrever um código excepcional.

Há muitas partes no processo de desenvolvimento de software relacionadas à criação de um trabalho de arte. O processo é:

Criativo

Ele exige imaginação. O software deve ser habilmente construído e o design deve ser preciso. Os programadores devem ter uma visão do código que estão prestes a criar e um plano de como eles o farão. Às vezes, isso envolve uma boa dose de engenhosidade.

Estético

Um bom código é marcado por elegância, beleza e equilíbrio. Ele se destaca no quadro de determinadas expressões culturais. Consideramos a forma do código, além de sua função.

Mecânico

Como qualquer artista, trabalhamos em nossos meios particulares, com nossas ferramentas, nossos processos e técnicas particulares. Trabalhamos com o patrocínio de benfeitores generosos.

Baseado em equipe

Muitas formas de arte não são empreendimentos de uma só pessoa.

> Nem toda forma de arte tem um artista sentado sozinho em seu estúdio, trabalhando dia e noite até que sua obra-prima esteja completa. Considere os mestres escultores com seus aprendizes. Considere a orquestra, com os membros formando um conjunto coordenado pelo maestro. Considere um compositor musical escrevendo uma peça que será então interpretada pelo(s) artista(s). Ou o arquiteto que projeta um prédio que será erguido por uma equipe de construtores.

Em vários aspectos, o conjunto de habilidades de um artista é semelhante ao de um programador.

Michelangelo foi o arquétipo do homem renascentista: foi pintor, escultor, arquiteto, poeta e engenheiro. Talvez ele tivesse se tornado um programador incrível. Quando lhe perguntaram como ele criou uma de suas obras mais famosas – a estátua de Davi –, ele disse: *olhei para a pedra e o vi lá; depois removi tudo o que estava sobrando*.

É isso que você faz? Você reduz e elimina as complexidades do espaço do problema, removendo tudo até atingir o belo código que você tinha em mente?

A seguir, listamos algumas perguntas para você fazer a si mesmo sobre o tema do software como uma arte:

- Eu considero os aspectos criativos do desenvolvimento de software ou eu o trato como uma atividade mecânica?
- Devo desenvolver um senso mais aguçado de elegância e de estética em meu código? Devo olhar para além do funcional e do que resolva o problema imediato?
- Eu acho que minha ideia de código "bonito" é a "única opinião verdadeira"? Devo considerar a arte como o objetivo de uma equipe?

Desenvolvimento de software é... uma ciência

Falamos de *ciência da computação*. Portanto deve haver algo vagamente científico em algum lugar, certo? Provavelmente, é razoável dizer que na maioria das empresas de desenvolvimento há muito menos ciência e

muito mais serviços de encanamento ocorrendo.

O arquétipo de cientista, é claro, é Albert Einstein. Ele não só foi um gênio como também é uma das pessoas cujas citações podem mais ser usadas (o que ajuda consideravelmente os autores). Ele afirmou que *qualquer tolo inteligente pode deixar as coisas maiores, mais complexas e mais violentas. É preciso um toque de gênio – e muita coragem – para se mover na direção oposta.*

Essa afirmação é realmente profunda: uma complexidade indevida acaba com a maioria dos projetos de software.

Einstein também era cultivador da estética. Apreciava a elegância e a beleza em suas teorias e tinha como meta reduzir os fatos a um todo coerente. Ele afirmou o seguinte: *sou artista suficiente para usar livremente a minha imaginação. A imaginação é mais importante que o conhecimento. O conhecimento é limitado. A imaginação dá volta ao mundo.*

Viu só? Eu disse que suas citações poderiam ser usadas.

Portanto, se o desenvolvimento de software é como uma ciência, o que isso quer dizer? Ele é (ou deveria ser):

Rigoroso

Procuramos ter um código livre de bugs, que funcione o tempo todo sempre. O código deve funcionar com todos os conjuntos de entradas válidas e deve responder apropriadamente a entradas inválidas. Um bom software deve ser preciso, comprovado, medido, testado e verificado.

Como conseguimos isso? Um bom teste é o segredo. Devemos ter testes de unidade, de integração e de sistema. De preferência, devemos ter testes automatizados para eliminar o risco de erro humano. Também procuramos realizar testes experimentais.

Sistemático

O desenvolvimento de software não é baseado em tentativa e erro. Você não pode ter como meta criar um sistema de computador grande e bem estruturado juntando componentes aleatórios de código até parecer que eles funcionem. É preciso planejar, criar um design, definir

um orçamento e desenvolvê-lo sistematicamente.

É um problema intelectual, lógico e racional; devemos trazer ordem e compreensão a partir do caos – que são o espaço do problema e as alternativas de design.

Esclarecedor

O desenvolvimento de software exige esforço intelectual e boas capacidades analíticas. Isso é especialmente evidente quando estivermos no encalço de bugs complicados. Como cientistas, criamos hipóteses e aplicamos algo semelhante a um método científico (formular uma hipótese, definir experimentos, executá-los e validar a teoria).

> **PONTO-CHAVE** Um bom desenvolvimento de software não é uma *codificação cowboy*, em que você usa o primeiro código que vier à sua mente. É um empreendimento planejado, avaliado e preciso.

Com base nisso, pergunte a si mesmo:

- Meu software sempre está totalmente correto e preciso? Como posso comprovar isso? Como posso deixar isso explícito, agora e no futuro?
- Eu me esforço para trazer ordem ao caos? Procuro reduzir a complexidade em meu código até que haja poucas partes pequenas e unificadas?
- Eu abordo os problemas de forma metódica e planejada ou corro de cabeça em direção a eles, de maneira não estruturada?

Desenvolvimento de software é... um esporte

A maioria dos esportes exige ótimas habilidades e esforço: tenacidade, treinamento, disciplina, trabalho em equipe, orientação e autoconsciência. Da mesma maneira, o desenvolvimento de software envolve:

Trabalho em equipe

Ele exige a coordenação entre várias pessoas com diferentes habilidades, trabalhando em harmonia.

Disciplina

Cada membro da equipe deve estar comprometido com ela e deve estar disposto a dar o melhor de si. Isso exige dedicação, trabalho árduo e muito treino.

Você não pode se tornar um bom jogador de futebol se ficar sentado no sofá assistindo a vídeos de treinos de futebol. Na verdade, se você fizer isso com algumas cervejas e um pacote de pipoca, é provável que vá piorar no futebol! Você deve realmente jogar, sair em campo com as pessoas e pôr em prática as suas habilidades; somente então irá melhorar. Você deve treinar – deve ter alguém que lhe diga como melhorar.

A equipe deve treinar junta e descobrir como poderá funcionar como um todo.

Regras

Jogamos (desenvolvemos) segundo um conjunto de regras e a cultura de uma equipe em particular. Isso está personificado em nossos processos e procedimentos de desenvolvimento, assim como nos ritos e nos rituais da equipe de software e nos fluxos de trabalho de suas ferramentas (considere o modo como há cooperação em torno de ferramentas como o sistema de controle de versões).

A analogia com o trabalho em equipe é mais evidente quando se trata de um esporte como o futebol. Você trabalha em um grupo de pessoas que atuam muito próximas umas das outras, jogando de acordo com um conjunto bem definido de regras.

Você já viu um time de garotos de sete anos jogando futebol? Um garotinho é deixado para trás, de pé, na boca do gol, e todas as demais crianças ficam correndo pelo campo perseguindo loucamente a bola. Não há passes. Não há comunicação. Ninguém está ciente dos demais membros do time. É somente um bando de crianças em direção a uma pequena esfera em movimento.

Compare isso com um time de alta qualidade da divisão principal. Esses times jogam de maneira muito mais coesa. Todos sabem qual é a sua

responsabilidade e o time trabalha em equipe, com coesão. Há uma visão compartilhada segundo a qual a equipe trabalha, e os jogadores formam um todo altamente funcional e bem coordenado.

- Eu tenho todas essas habilidades? Trabalho bem em equipe ou posso melhorar em algumas áreas?
- Estou comprometido com minha equipe, me dispondo a trabalhar pelo bem de todos?
- Continuo aprendendo sobre desenvolvimento de software? Aprendo com os outros e estou aperfeiçoando minhas habilidades para trabalhar em equipe?

Desenvolvimento de software é... uma brincadeira de criança

Para mim, essa observação parece ser particularmente apropriada: no fundo, sou apenas uma criança. Não é o que todos somos?

É fascinante ver como as crianças crescem e aprendem, como sua visão de mundo muda e é moldada a cada nova experiência. Podemos aprender muito com a maneira de uma criança conhecer e reagir ao mundo.

Considere como isso se aplica ao nosso desenvolvimento de software:

Aprendizado

Uma criança está ciente de que está aprendendo, de que não sabe tudo. Isso exige uma característica simples: *humildade*. Alguns dos programadores com que tive mais dificuldade de trabalhar achavam que sabiam tudo. Se houvesse algo novo que tivessem de aprender, eles liam um livro e presumiam que eram *experts* no assunto. A humildade passava longe.

Uma criança está constantemente assimilando novos conhecimentos. Devemos reconhecer que, se quisermos melhorar, devemos aprender. E devemos ser realistas sobre o que sabemos e o que não sabemos.

Aprecie o aprendizado, saboreie a descoberta de novidades. Pratique e aperfeiçoe a sua arte.

PONTO-CHAVE Bons programadores trabalham com humildade. Eles admitem que

não sabem tudo.

Simplicidade

Você escreve o código mais simples possível? Reduz tudo à forma menos complexa possível para facilitar a compreensão e a codificação?

Amo o modo como as crianças tentam chegar ao fundo de uma questão para entender tudo a partir de suas perspectivas limitadas. Elas estão sempre perguntando *por quê*. Considere, por exemplo, uma conversa que tive com minha filha quando ela tinha seis anos. *Papai, por que Millie é minha irmã?* Porque você está na mesma família que ela, Alice. *Por quê?* Bem, porque você tem a mesma mamãe e o mesmo papai. *Por quê?* Porque, veja bem, existem pássaros e abelhas... Ah, vá ler um livro! *... (pensando) … Por quê?…*

Nós devíamos estar constantemente perguntando "*por quê?*" – questionando o que estamos fazendo e os motivos pelos quais estamos fazendo; procurando ter uma melhor compreensão do problema e buscando a melhor solução. E devemos nos esforçar para ter simplicidade em nossos trabalhos. Isso não quer dizer criar um código "tolo" *simplista*, porém um código não complexo que seja adequado.

Diversão

Se tudo o mais falhar, não haverá nada de errado com isso. Todos os bons desenvolvedores apreciam um pouco de diversão. Na empresa em que trabalho, atualmente, temos um monociclo e um campo de críquete improvisado.

Com isso em mente, podemos nos perguntar se:

- Eu me esforço para escrever o código mais simples possível? Ou digito o que vier à minha mente e não penso em tarefas comuns, refatoração ou design de código?
- Eu continuo aprendendo? Sobre o que eu posso aprender? Sobre o que eu devo aprender?
- Sou um programador humilde?

Desenvolvimento de software é... uma obrigação

Muito de nosso trabalho de desenvolvimento de software não é agradável. Não é glamoroso. Não é um mar de águas tranquilas. É somente um trabalho de burro de carga que deve ser feito para que o projeto seja concluído.

Para ser um programador eficiente, você não deve ter medo das obrigações. Reconheça que a programação *é* um trabalho árduo. Sim, é muito bom criar um ótimo design na versão mais recente do produto, porém, às vezes, é preciso fazer correções tediosas de bugs e vasculhar o código antigo e confuso para conseguir que um produto seja lançado e ganhar dinheiro.

De vez em quando, devemos nos tornar zeladores do software. Isso exige:

Fazer limpeza

Devemos identificar os problemas e corrigi-los; descubra onde estão as falhas e quais são as correções apropriadas. Essas correções devem ser feitas no prazo e de modo a não desestruturar o código. Um zelador não deixa as tarefas desagradáveis para outras pessoas, mas assume a responsabilidade por elas.

Trabalhar em segundo plano

Os zeladores não trabalham no centro das atenções. Provavelmente, são pouco reconhecidos pelos seus esforços heroicos. É muito mais uma função de suporte, e não de liderança.

Fazer manutenção

Um zelador de software removerá códigos mortos, corrigirá códigos que não estejam funcionando, fará refatorações e reimplementações de trabalhos inadequados, organizará e limpará o código para garantir que ele não caia em ruínas.

Pergunte a si mesmo:

- Fico satisfeito em realizar as *obrigações* no código? Ou eu só quero fazer o trabalho glamoroso?
- Eu assumo a responsabilidade por códigos confusos e faço uma

limpeza?

Excesso de metáforas

Com frequência, criamos metáforas para o ato de desenvolver software. Muitas das ideias que reunimos podem ser informativas. Entretanto nenhuma metáfora é perfeita. O desenvolvimento de software é especial por si só, e o ato de criá-lo não é totalmente igual a nenhuma outra disciplina. É um campo que continuamos explorando e refinando. Tome cuidado para não realizar deduções falsas a partir de comparações ruins.

Um bom código e bons programadores nascem do desejo de escrever o que é certo da maneira certa, e não do equivalente de software ao creme de letrinhas.

Perguntas

1. Com quais das metáforas apresentadas aqui você se relaciona mais claramente? Qual delas reflete melhor o seu trabalho atual?
2. Que outras metáforas você pode criar para a criação de software? (Talvez com jardinagem ou pastoreio.) Que novas ideias elas revelam?
3. Como *você* faria um creme de letrinhas?

Veja também

- *Importar-se com o código* (Capítulo 1) – Devemos nos importar com a criação de um *software correto* da *maneira correta*.
- *O poder das pessoas* (Capítulo 34) – Contém algumas metáforas relacionadas ao trabalho de desenvolvimento de software em equipe. Programação é uma meta para pessoas.

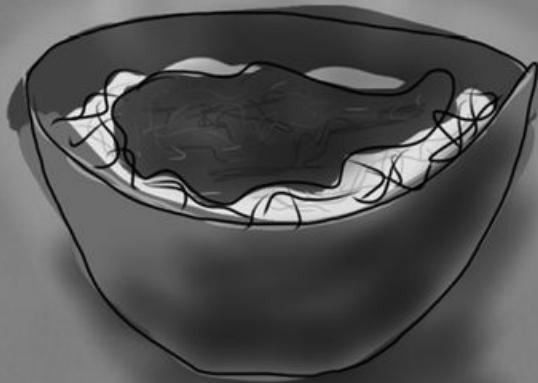
TENTE ISTO... Retorne às perguntas anteriores. Em quais áreas você deve focar mais nesse momento?

10.000 MACACOS

(OU ALGO POR AÍ)

TRABALHANDO COM SOFTWARE

NENHUM
(É UM PROBLEMA DE HARDWARE)

```
for (socket in ceilingFixings)
{
  Bulb bulb = socket.getBulb();
  if (!bulb) continue; // Estava pensando por que estava
                       // escuro aqui
  bulb.unScrew();
  if (bulb.shattered())
    throw handsInTheAir(); // ai
  bin.accept(bulb); // TODO: reciclar
  Bulb replacement = new Bulb;
  socket.accept(replacement);
  if (!replacement.fits(socket))
    replacement.pushItHarder();
  run.away();
}
```

(NA VERDADE, SÓ PRECISAMOS DE UM, MAS ELE DEVE LER TODO O MANUAL DE INSTRUÇÕES ANTES)

---

1 N.T.: *craft*, em inglês.

CAPÍTULO 15

Jogando segundo as regras

Se eu tivesse obedecido a todas as regras, jamais teria chegado a lugar algum.

– Marilyn Monroe

Vivemos nossas vidas segundo várias regras. Isso poderia ser um pesadelo distópico de George Orwell, mas não é. Algumas regras nos são impostas. Contudo outras são definidas por nós mesmos. Essas regras lubrificam as engrenagens de nossas vidas.

As regras facilitam o nosso jogo, descrevendo como ele funciona: elas

dizem quem ganhou e como. Tornam nossos esportes justos e aprazíveis, além de oferecer muitas oportunidades para (erros de) interpretação (veja a regra de impedimento no futebol).

Elas causam impacto em nossas viagens, em que as regras de segurança determinam que você pode levar uma quantidade limitada de líquido e não pode carregar nenhum objeto pontiagudo nos aviões. Descrevem os limites de velocidade no trânsito e como transitar de modo seguro em uma estrada. Essas regras garantem a segurança de todos.

As regras estão vinculadas às nossas normas sociais, determinando que não é apropriado lamber a orelha de um desconhecido tão logo você o conheça, independentemente do quão apetitosa ela lhe parecer.

Sim, vivemos nossas vidas observando continuamente um conjunto de regras. Estamos tão acostumados com isso que, com frequência, não pensamos no assunto.

Não é de surpreender que o mesmo ocorra em nosso trabalho de desenvolvimento. Há uma enorme variedade de regras que seguimos para o código. Normas para os processos de desenvolvimento. Conjuntos de ferramentas e fluxos de trabalho obrigatórios. Etiqueta no ambiente de trabalho. Sintaxe da linguagem. Padrões de projeto. São os aspectos que definem o que é ser um programador profissional e o modo como participamos do jogo do desenvolvimento com outras pessoas.

Ao se juntar a um novo projeto, haverá diversas regras que você poderá esperar que estejam definidas. Regras que governam a criação responsável de um código de alta qualidade. Regras que governam os processos e as práticas do trabalho. E regras específicas sobre o projeto e o domínio do problema: talvez regulamentações legais em vigor para negócios financeiros ou diretrizes de segurança para os mercados de saúde.

Essas regras nos ajudam a trabalhar bem juntos. Ajudam a coordenar e a harmonizar os nossos esforços.

Precisamos de mais regras!

Às vezes, porém, todas essas regras, por melhores que sejam, não são

suficientes. De vez em quando, os pobres programadores precisam de *mais* regras. Realmente, nós precisamos.

Precisamos de regras *criadas por nós mesmos*. Regras das quais podemos ser os donos. Regras que definam a cultura e os métodos de trabalho de nossa equipe em particular para o desenvolvimento. Essas não precisam ser editais draconianos enormes e pesados. Basta ser algo simples, que possa ser dado aos novos membros da equipe para que eles possam participar imediatamente do jogo com você. São regras que descrevem algo mais do que simples métodos e processos; são regras que descrevem uma cultura de codificação – como ser um bom jogador na equipe.

PONTO-CHAVE As equipes de programação têm um conjunto de regras. Essas regras definem *o que* fazemos e *como* fazemos. Mas elas também descrevem uma *cultura de codificação*.

Parece razoável? Bem, nós achamos que sim. O *Tao* de nossa equipe para o desenvolvimento está sintetizado em três breves declarações complementares. Todas as demais práticas se originam a partir daí. Essas declarações estão consagradas no folclore de nossa equipe, foram impressas em cartazes grandes e simpáticos e são exibidas nas áreas de trabalho comuns. Elas reinam sobre tudo o que fazemos; sempre que estamos diante de uma escolha, uma decisão complicada ou em uma discussão acalorada, elas nos guiam em direção à resposta correta.

Você está pronto para compartilhar de nossa sabedoria? Prepare-se. Nossas três impressionantes regras para escrever um bom código são:

- mantenha a simplicidade;
- use o seu cérebro;
- nada está gravado a ferro e fogo.

É isso. Elas não são ótimas?

Definimos essas regras porque achamos que elas nos conduzem a um software melhor e têm nos ajudado a nos tornar melhores programadores. Vou descrever o que elas querem dizer nos próximos capítulos.

Elas descrevem perfeitamente a atitude, o senso de comunidade e a cultura de nossa equipe. Nossas regras são propositadamente breves e

expressivas; não gostamos de ditados longos e burocráticos nem de complicações desnecessárias. Elas exigem a responsabilidade do desenvolvedor para serem interpretadas e seguidas; confiamos em nossa equipe e essas regras conferem poder a ela. Sempre há novas maneiras de aplicá-las em nossa base de código; estamos sempre aprendendo e procurando melhorar.

Defina as regras

Essas regras fazem sentido para nós, em nosso projeto, em nossa empresa e em nosso mercado. Pode ser que elas não tenham a mesma importância para você.

De acordo com quais regras você está trabalhando no momento? Quero dizer, além daquela que proíbe lamber as orelhas de seus colegas. Você tem um padrão de codificação (seja formal ou informal) definido? Tem regras para processos de desenvolvimento (talvez do tipo: *Esteja presente às dez da manhã porque temos uma reunião diária*. *Todo código deve ser revisado antes de um check-in. Todos os relatórios de bug devem ter passos claros para a sua reprodução antes de serem passados para um desenvolvedor*)?

Quais regras governam a cultura de sua equipe? Quais modos de cooperação informais e não registrados por escrito, ou quais abordagens ao código, são particulares à sua equipe?

Considere formular um conjunto pequeno e simples de regras segundo as quais você possa definir a sua cultura de codificação. É possível reduzi-las a algo mais conciso e expressivo como no caso de nossas três regras?

> **PONTO-CHAVE** Não dependa de “regras” vagas e não registradas por escrito para a equipe. Torne explícitas as regras implícitas e assuma o controle de sua cultura de codificação.

No espírito de nossa terceira regra, não se esqueça de que *nada está gravado a ferro e fogo* – inclusive as suas regras. Afinal de contas, as regras existem para serem infringidas. Ou melhor, elas existem para serem *refeitas*. Suas regras podem mudar com o passar do tempo, de modo

justificável, à medida que sua equipe aprender e crescer. O que é pertinente agora poderá não ser no futuro.

Perguntas

1. Liste as regras do processo de desenvolvimento de software definidas atualmente para o seu projeto. Como elas são aplicadas e seguidas?
2. Como a cultura desse projeto difere da cultura de projetos anteriores? É um projeto melhor ou pior para trabalhar? A diferença pode ser identificada ou melhorada e se transformar em uma regra?
3. Você acha que sua equipe pode se unir em torno de um conjunto de regras sobre o qual haja consenso?
4. O formato, o estilo e a qualidade de seu código têm algum efeito sobre a cultura de codificação de um projeto? A equipe molda o código ou é o código que molda a equipe?

Veja também

- *Mantenha a simplicidade* (Capítulo 16), *Use o seu cérebro* (Capítulo 17), *Nada está gravado a ferro e fogo* (Capítulo 18) – Exposição das três incríveis regras de minha equipe para um desenvolvimento de software eficiente.
- *Manifestos* (Capítulo 37) – O lado militante da criação de regras – os manifestos.
- *É a ideia que vale* (Capítulo 35) – Deve haver consenso entre você e as demais pessoas para que as regras da equipe sejam seguidas e você deve ser responsável por essas regras.

TENTE ISTO... Formule as "regras" de desenvolvimento de software para a sua própria equipe. Imprima-as e cole-as em uma parede de seu escritório de desenvolvimento.

10.000 MACACOS
(OU ALGO POR AÍ)
AS LEIS DO DESENVOLVIMENTO DE SOFTWARE
A LEI DO ACESSO INDIRETO
TODO PROBLEMA PODE SER RESOLVIDO PELA ADIÇÃO DE UM NÍVEL EXTRA DE ACESSO INDIRETO
DAVID WHEELER
10,000 MONKEYS
SOFTWARE DEVELOPMENT
ESTÁ TUDO EXPLICADO AQUI!
LEI DE BROOKE
ADICIONAR PESSOAS A UM PROJETO DE SOFTWARE ATRASADO SOMENTE IRÁ ATRASÁ-LO
FRED BROOKES
VOCÊ GOSTARIA DE UMA AJUDA?
A LEI 90/90
VOCÊ ESCREVE OS PRIMEIROS 90% DE CÓDIGO NOS PRIMEIROS 90% DO TEMPO DE DESENVOLVIMENTO. OS 10% DE CÓDIGO RESTANTE CONSOMEM OS OUTROS 90% DO TEMPO DE DESENVOLVIMENTO
TOM CARGILL
TEMPO EXTRA INDIRETO

CAPÍTULO 16
Mantenha a simplicidade

A simplicidade é o último grau da sofisticação.

– Leonardo da Vinci

Você já ouviu o conselho antes: "KISS". *Keep It Simple, Stupid* (Mantenha a simplicidade, seu estúpido). Exatamente, que nível de estupidez você deve ter para não entender *isso*? A simplicidade, sem dúvida, é um excelente objetivo; com certeza, você deve se esforçar para tê-la em seu código. Nenhum programador anseia trabalhar com um código excessivamente complexo. Um código simples é transparente; sua estrutura é clara, ele não oculta bugs, é simples de aprender e é fácil trabalhar com ele.

Então por que nem todos os códigos são assim?

No mundo dos desenvolvedores, há dois tipos de simplicidade: o tipo *errado* e o tipo *certo*. A "simplicidade" que estamos buscando *não* quer dizer especificamente: escrever o seu código da maneira mais fácil que puder, usar atalhos, ignorar tudo o que for desagradavelmente complicado (varrer tudo para baixo do tapete e esperar que desapareça) e, de modo geral, ser simplório quanto à programação.

Ah, se fosse tão fácil assim... Muitos programadores no mundo real escrevem códigos "simples" dessa maneira. Seus cérebros não são acionados. Alguns deles nem sequer percebem que estão fazendo algo errado: eles simplesmente não pensam o suficiente sobre o código que escrevem e deixam de perceber todas as complexidades sutis que lhe são inerentes.

Uma abordagem descuidada como essa não resulta em um código simples, mas em um código *simplista*. Um código simplista é um código incorreto. Como não é bem planejado, ele não executa exatamente

conforme necessário – com frequência, o código inclui somente o "caso principal" óbvio, ignorando as condições de erro, ou não trata corretamente as entradas menos prováveis. Por esse motivo, um código simplista abriga falhas. São fissuras que (da maneira típica de um programador simplista) tendem a ser cobertas com mais código simplista indevidamente aplicado. Essas correções começam a se empilhar umas sobre as outras até o código se transformar em uma confusão monstruosa; é exatamente o oposto de um código simples e bem estruturado.

A simplicidade jamais deve ser uma desculpa para um código incorreto.

> **PONTO-CHAVE** Criar o design de um código simples exige esforço. Não é o mesmo que criar um código excessivamente *simplista*.

No lugar dessa "simplicidade" ingênua e incorreta, devemos nos esforçar para criar o código *mais simples* possível. Isso é muito diferente de desengatar o seu cérebro e escrever um código estúpido e simplista. É uma meta para um cérebro em intenso funcionamento – ironicamente, é difícil escrever algo simples.

Designs simples

Há um sinal incontestável de um design simples: o fato de ele poder ser descrito de forma rápida e clara e poder ser facilmente compreendido. Você pode resumi-lo em uma frase simples ou em um diagrama claro. Designs simples são fáceis de conceituar.

Designs simples têm várias propriedades dignas de nota. Vamos dar uma olhada nelas.

Simples de usar

Um design simples é, por definição, fácil de usar. Não exige um nível excessivamente elevado de cognição.

É fácil de compreender porque não há muito que aprender no início. Você pode começar a trabalhar com as funcionalidades mais básicas, e, à medida que for necessário adotar recursos mais avançados, esses se

revelarão gradualmente, como em uma narrativa bem redigida.

Evita usos indevidos

É difícil usar indevidamente um design simples e é difícil abusarmos dele. Esse tipo de design reduz a carga para os clientes do código ao manter interfaces claras e evitar a imposição de uma carga desnecessária no usuário. Por exemplo, um design "simples" de interface não retornará objetos alocados dinamicamente, que o usuário deverá apagar manualmente. O usuário irá se esquecer. Haverá leak ou falhas no código.

O segredo consiste em colocar a complexidade nos lugares certos: em geral, ela deve estar oculta, por trás de uma API simples.

> **PONTO-CHAVE** Designs simples têm como objetivo evitar usos indevidos. Eles podem envolver uma complexidade interna extra para que uma API mais simples seja apresentada.

O tamanho importa

Um código simples minimiza a quantidade de componentes no design. Projetos grandes, com muitas partes, podem exigir uma grande quantidade de componentes, de modo justificável; é possível ter várias partes separadas e ser "o mais simples possível".

> **PONTO-CHAVE** Designs simples são tão pequenos quanto possíveis. E não menores.

Caminhos mais curtos no código

Você se lembra da famosa máxima dos programadores: *todo problema pode ser resolvido pela adição de um nível extra de acesso indireto*? Muitos problemas complexos podem ser sutilmente mascarados e até mesmo causados por um nível extra de acesso indireto e desnecessário que oculte o problema. Se você tiver de seguir uma cadeia longa de chamadas de funções ou seguir acessos indiretos a dados por meio de muitos níveis de funções "getter", sistemas de encaminhamento e níveis de abstração, você logo perderá a vontade de viver. É desumano. É desnecessariamente complexo.

Designs simples reduzem os níveis de acessos indiretos e garantem que a funcionalidade e os dados estejam próximos dos locais em que forem necessários.

Eles também evitam herança, ligações dinâmicas ou polimorfismo desnecessários. Essas técnicas são todas boas quando usadas nos momentos certos. Porém, quando aplicadas cegamente, podem gerar uma complexidade desnecessária.

Estabilidade

O sinal indubitável de um design simples é que ele pode ser melhorado e estendido sem quantidades massivas de reescrita. Se você acabar retrabalhando continuamente uma seção de código à medida que seu projeto amadurecer, é sinal de que você tem um conjunto de requisitos ridiculamente volátil (o que *pode* acontecer, porém é um problema bem diferente) ou é uma indicação de que o design não é simples o suficiente, antes de tudo.

Interfaces simples tendem a ser estáveis e não mudam muito. Você pode estendê-las com novos serviços, porém não é necessário refazer toda a API. Entretanto isso não deve ser uma camisa de força: as interfaces não precisam estar gravadas a ferro e fogo. Não torne o seu código desnecessariamente rígido – isso, por si só, não é ser simples.

Linhas de código simples

Um código simples é fácil de ler e de entender. Portanto é fácil trabalhar com ele.

Preferências pessoais e familiaridade tendem a determinar o que fazem com que linhas individuais de código pareçam ser simples. Algumas pessoas acham que determinados idioms de layout ajudam a tornar seus códigos mais claros. Outras acham que esses mesmos idioms são um obstáculo enorme. Acima de tudo, a *consistência* resulta em um código simples. Um código com estilos, convenções de nomenclatura, abordagens de design e formatos de arquivo que variem muito será um código desnecessariamente obscuro.

PONTO-CHAVE A consistência leva à clareza.

Não escreva códigos desnecessariamente obscuros, qualquer que seja o motivo: não para assegurar o seu emprego (fazemos piada a esse respeito, mas algumas pessoas realmente fazem isso), não para impressionar seus colegas com a sua maestria na codificação e não para experimentar um novo recurso da linguagem. Se puder criar uma implementação aceitável com um estilo de codificação comum, porém claro, faça isso. Os programadores responsáveis pela manutenção agradecerão.

Mantenha a simplicidade e não a estupidez

Caso encontre um bug, saiba que normalmente há duas maneiras de abordá-lo:

- Tomar o caminho mais fácil para solucionar o problema. Ei, você está mantendo a *simplicidade*, certo? Corrija o problema superficial – ou seja, aplique um esparadrapo –, mas não se preocupe em resolver quaisquer problemas subjacentes mais profundos se der muito trabalho. É um esforço mínimo de sua parte agora, porém é provável que isso vá conduzi-lo ao tipo de código simplista e confuso que vimos anteriormente.

  Isso não torna o código mais simples; ele se tornará mais complexo. Uma nova imperfeição foi acrescentada e o problema subjacente não foi tratado.

- Ou você pode retrabalhar o código para que ele acomode uma correção e permaneça simples. Pode ser que seja necessário ajustar APIs para que elas se tornem mais adequadas, refatorar uma lógica para efetuar a correção apropriada para o bug ou até mesmo realizar um retrabalho sério porque você identificou suposições no código que não se sustentam.

Essa última opção é a meta. Ela exige mais esforços, porém reduzir o código à sua forma mais simples compensa no longo prazo.

PONTO-CHAVE Aplique correções de bugs à causa-raiz, e não nos locais em que os

sintomas se manifestam. Correções feitas com esparadrapos nos sintomas não resultam em códigos simples.

As suposições podem reduzir a simplicidade

Suposições "simplificadoras" inválidas são fáceis de fazer quando você estiver codificando e, embora possam reduzir a complexidade em sua mente, elas tendem a se transformar em uma lógica complicada.

Um código simples não faz suposições desnecessárias, seja sobre os requisitos ou sobre o domínio do problema, sobre o leitor, o ambiente de execução ou o conjunto de ferramentas usado. As suposições podem reduzir a simplicidade, pois exigimos implicitamente que o leitor tenha informações extras para entender o código.

PONTO-CHAVE Evite suposições implícitas em seu código.

As suposições, porém, podem *aumentar* a simplicidade. O truque é tornar claras quais suposições exatamente estão sendo feitas; por exemplo, as restrições e o contexto para os quais o código está sendo projetado.

Evite uma otimização prematura

A otimização é a antítese da simplicidade. Knuth tem uma frase famosa que diz que *uma otimização prematura é a causa de todo o mal (ou pelo menos, da maior parte dele) na programação*.[1]

O ato de otimizar o código, em geral, consiste em tomar uma implementação direta, legível e algorítmica e estraçalhá-la: acabar com a forma do algoritmo para que ele execute mais rapidamente em um determinado computador em determinadas condições. Isso, inevitavelmente, altera o formato de modo que fique menos claro e, sendo assim, menos simples.

Implemente um código claro inicialmente. Torne-o complexo somente quando for necessário.

Empregue uma ordenação simples e um padrão até que tenha de deixá-la mais inteligente. Faça a implementação mais simples de um algoritmo e,

em seguida, avalie para saber se ela deverá ser mais rápida. Novamente, tome cuidado ao fazer suposições: muitos programadores otimizam as partes que eles *acham* que serão lentas. Os gargalos geralmente estão em outros lugares.

Suficientemente simples

A simplicidade é aliada da *suficiência*. Isso funciona em algumas direções:

- Você deve trabalhar da maneira mais simples possível e criar o código mais simples possível. Porém mantenha-o suficientemente simples. Se você simplificá-lo demais, o problema em questão não será solucionado. Nossas soluções “simples” *devem* ser “suficientes”; do contrário, elas *não* serão soluções.
- Escreva somente a quantidade de código necessária para resolver o seu problema. Não escreva pilhas de código que você *ache* que serão úteis. Um código que não estiver em uso será somente uma bagagem. Será um peso extra. É uma complexidade de que você não precisará. Escreva a quantidade suficiente de código. Quanto menos código você escrever, menos bugs serão criados.
- Não complique demais as soluções; desenvolvedores empolgados acham isso realmente tentador. Resolva somente o problema que você tiver em mãos. Não invente uma solução geral desnecessariamente para todo um conjunto de problemas que não sejam relevantes. Trabalhe até ter alcançado uma bela suficiência.

PONTO-CHAVE Escreva somente a quantidade de código necessária. Tudo o mais será uma complexidade que se tornará um peso.

Uma conclusão simples

Todos nós sabemos que um código maravilhosamente simples é melhor que um código desnecessariamente complexo. E todos nós já vimos nossa quota de código abominável, feio e complexo. Poucas pessoas se propõem a criar um código como esse. O caminho para a complexidade geralmente é marcado por mudanças apressadas e padrões que deixaram de ser

usados. É apenas uma mudança descuidada. É apenas uma correção feita com esparadrapos. É apenas uma revisão de código que deixou de ser feita. É apenas um “não tenho tempo para refatorar”. Depois de vários desses, o código se transformará em uma enorme confusão e será difícil achar uma maneira de restaurar a sanidade.

Infelizmente, simplicidade significa trabalho árduo.

A simplicidade é uma bandeira surgida de várias máximas populares dos desenvolvedores: YAGNI – *You Aren’t Going to Need It* (Você não precisará disso) – aborda o tema da suficiência; DRY – *Don’t Repeat Yourself* (Não se repita) – relaciona-se ao tema do tamanho do código. Nossa preferência por alta coesão e baixo acoplamento relaciona-se com a simplicidade no design.

Perguntas

1. Qual foi o código mais simples que você viu recentemente? Qual foi o código mais complexo que você viu? Quais são as diferenças entre eles?
2. Que tipos de suposições desnecessárias um programador pode fazer sobre o seu código, que o levará a ser complexo demais? Quais suposições são válidas?
3. Falamos bastante sobre otimização no nível do código. Como é possível otimizar nos níveis de design e de arquitetura?
4. É possível otimizar o código, porém manter a sua simplicidade?
5. A “simplicidade“ de uma seção de código depende das habilidades do programador que a estiver lendo? Como um programador experiente deve trabalhar para garantir que seu código seja de alta qualidade, porém pareça “simples” para um programador menos experiente, responsável pela manutenção do código?

Veja também

- *Jogando segundo as regras* (Capítulo 15) – “Mantenha a simplicidade” é uma das três regras complementares criada pela minha equipe.
- *Lidando com a complexidade* (Capítulo 12) – O inverso da simplicidade

é a *complexidade*. Esse capítulo descreve como administrá-la.

> **TENTE ISTO...** Verifique se as modificações de código que você está fazendo contribuem para a simplicidade do código. Evite adicionar complexidade. Lute contra a entropia no código!

---

1 Em *Computer Programming as an Art* (Programação de computadores como uma arte) – sua palestra no Turing Award em 1974.

CAPÍTULO 17

Use o seu cérebro

"Abel é inteligente", disse o Ursinho Puff pensativo.
"Sim", disse Leitão, "Abel é inteligente".
"E ele tem cérebro."
"Sim", disse Leitão, "Abel tem cérebro".
Houve um longo silêncio. "Acho", disse o Ursinho Puff,
"que é por isso que ele nunca entende nada".

– A. A. Milne
O Ursinho Puff

"Use o seu cérebro" não é uma ordem depreciativa para colegas negligentes. Pelo contrário, é um princípio essencial para o programador consciente. É a segunda das regras escolhidas pela minha equipe de gurus da programação. Ela tem diversas aplicações importantes em nosso regime diário de codificação.

Não seja estúpido

Já mencionamos a regra KISS: *Keep It Simple, Stupid* (Mantenha a simplicidade, seu estúpido). Vamos dar um passo além aqui: *não seja estúpido*. Parece um conselho óbvio, porém nós, programadores, precisamos de lembretes repetitivos.

É incrível como pessoas superinteligentes podem ser tolas. Alguns gênios sofrem de ausência crônica da glândula do bom senso. São ninjas que tropeçam em seus códigos por causa de sua visão míope; cegamente, eles deixam de ver o óbvio bem à sua frente. Arquitetos incríveis batem nas paredes porque suas cabeças estão enfiadas nas nuvens.

É típico dos geeks.

O desejo de criar um novo algoritmo empolgante ou de compor uma estrutura de dados ardilosa pode nos consumir, obscurecendo o fato de que um simples array seria suficiente. Na pressa de disponibilizar uma versão, é fácil gerar uma imensidão de códigos abaixo do padrão; a pressão faz com que fiquemos tentados a pensar com menos cuidado. Criamos um código *estúpido*.

Especialistas em codificação fazem isso, e nós, meros mortais, podemos fazer o mesmo. Certifique-se de que seu código não irá cegamente deixar de lado o óbvio. Não complique demais os designs acidentalmente. Não adicione uma estupidez que poderia ser facilmente evitada.

> **PONTO-CHAVE** Pare e pense. Não escreva códigos estúpidos.

Entretanto todos nós cometemos erros de vez em quando. Ninguém escreve códigos perfeitos consistentemente. Portanto não se sinta paralisado nem pense que você seja um fracassado quando perceber que escreveu algum código estúpido ou que criou um design idiota.

Simplesmente admita quando estiver errado, volte atrás em seu trabalho e adote uma abordagem melhor. Admitir uma falha e retrabalhar o erro exige coragem. É mais corajoso fazer isso do que tentar manter a compostura e tentar usar um código defeituoso. Trate o código com respeito. Limpe a sua sujeira.

> **PONTO-CHAVE** Admita seus erros e as decisões ruins de codificação. Aprenda com eles.

Evite o descuido

Seja honesto, todos nós já fizemos isto: programamos no piloto automático.

É muito fácil programar sem envolver o seu cérebro. De verdade. É fácil simplesmente seguir seus dedos enquanto eles digitam linhas de código. É fácil ficar preso a uma trilha tentando solucionar (o que você acha ser) o problema imediato, sem realmente considerar o quadro geral, sem pensar no código ao seu redor ou se o que você está digitando realmente está

correto.

Inevitavelmente, isso resulta em um código estúpido. Resulta em um código extenso e complexo demais. Resulta em um código incorreto, que não atenderá a todos os requisitos. Resulta em um código com bugs, que não tratará todos os casos.

Sempre que estiver diante de uma tarefa de codificação, pare, dê um passo mentalmente para trás e considere se não há uma solução alternativa. Verifique se você não está trabalhando com o primeiro plano que surgiu em sua mente só porque você não tentou pensar em alternativas.

Para parafrasear a propaganda da Primeira Guerra Mundial, *um código descuidado custa vidas*.

> **PONTO-CHAVE** Preste atenção. Não escreva códigos de forma descuidada.

As melhores estratégias para evitar a armadilha do descuido e a sua própria estupidez envolvem responsabilidade. Faça revisões de design antes de mergulhar em seu editor. Faça a programação aos pares. Execute revisões de código.

Você tem permissão para pensar!

"Use o seu cérebro", acima de tudo, é uma regra que atribui poder. Você realmente tem *permissão* e é até mesmo incentivado a usar o seu cérebro.

Alguns programadores falham em assumir responsabilidade suficiente. Eles trabalham como macacos criando códigos, preenchendo as lacunas dos designs de outras pessoas ou seguindo estruturas e idioms existentes, em vez de terem poder para pensar por conta própria.

Você não é um autômato codificando. Você tem um cérebro: use-o!

À medida que trabalhar em uma seção de código, tome decisões conscientes sobre o seu formato e a estrutura. *Seja dono* do código. Assuma a responsabilidade por ele. Seja pró-ativo para determinar qualquer melhoria ou mudança necessária.

Se os padrões existentes do código forem questionáveis, considere se eles devem ser alterados. Faça uma avaliação para saber se agora é o momento

certo de refatorar.

Se você achar o código repleto de esparadrapos, não siga em frente e adicione outro quando um ajuste mais radical for necessário. Entenda que identificar esse tipo de problema é *sua responsabilidade*. Você tem permissão para avaliar o código de forma crítica.

Ter uma opinião e sentir-se capaz de emiti-la exige que você seja bravo e corajoso. Defenda aquilo que vá trazer melhorias ao código.

> **PONTO-CHAVE** Tenha coragem de usar o seu cérebro. Sinta que você tem autonomia para criticar o código e tomar decisões sobre como melhorá-lo.

Perguntas

1. Qual é a diferença entre um código *simples* e um código *estúpido*?
2. Como você pode garantir que não criará um código estúpido? Você acha que tem "bom senso" para criar um bom código? Justifique sua resposta.
3. Quais são os sinais evidentes de que o código foi escrito por alguém que não estava prestando atenção?
4. Quais são os fatores decisivos para optar entre retrabalhar uma seção de código ruim ou marcá-la "pragmaticamente" como débito técnico e dar o fora?

Veja também

- *Jogando segundo as regras* (Capítulo 15) – "Use o seu cérebro" é uma das três regras complementares criadas pela minha equipe.
- *Dessa vez, eu entendi* (Capítulo 33) – Um caso de estudo sobre quando dar um passo para trás e usar a massa cinzenta.

> **TENTE ISTO...** Preste mais atenção quando estiver trabalhando. Neste momento, selecione duas técnicas que ajudarão você a focar melhor e a evitar a criação de um código *estúpido*.

10.000 MACACOS
(OU ALGO POR AÍ)
O PODER DO CÉREBRO
O CÉREBRO É UMA MÁQUINA MARAVILHOSA
É O COMPUTADOR MAIS EFICAZ QUE CONHECEMOS
CONTÉM CEM BILHÕES DE NEURÔNIOS
TEM UMA CAPACIDADE INCRÍVEL DE LEMBRAR FATOS DISTANTES
EXIGE CERCA DE 20% DOS RECURSOS DE SEU CORPO PARA FUNCIONAR
É UMA MÁQUINA QUÍMICA COMPLEXA PARA PROCESSAMENTO E ARMAZENAMENTO
É UM BANCO DE DADOS AUTOMÁTICO DE REFERÊNCIAS CRUZADAS
APESAR DISSO, NÃO CONSIGO ME LEMBRAR ONDE COLOQUEI MINHAS CHAVES ONTEM À NOITE

CAPÍTULO 18
Nada está gravado a ferro e fogo

Sempre dizem que o tempo muda as coisas, mas, na verdade, é você quem deve mudá-las.

– Andy Warhol

Há uma estranha crença dominante nos círculos de programação: depois que um código é escrito, ele se torna sagrado. Ele não deve ser alterado. Jamais.

Isso vale o dobro para o código de outra pessoa. Não ouse tocá-lo.

Em algum ponto durante o desenvolvimento, talvez no primeiro check-in ou logo depois que um produto é disponibilizado, o código se torna embalsamado. Ele muda de nível. É promovido. Não pertence mais à plebe; ele passa a fazer parte da realeza digital. O design, anteriormente questionável, repentinamente passa a estar além de qualquer censura e se torna inalterável. Ninguém deve mais mexer com a estrutura interna do código. Todas as interfaces com o mundo externo se tornam sagradas e jamais deverão ser revisadas.

Por que os programadores *pensam* dessa maneira? Medo. Medo de entender errado. Medo de introduzir falhas. Medo de trabalho adicional. Medo do custo da alteração.

Há uma verdadeira ansiedade resultante de mudar um código que você não entenda completamente. Se não entender toda a lógica, se não estiver totalmente certo do que está fazendo, se não souber quais são todas as possíveis consequências de uma mudança, você *poderá* fazer o programa deixar de funcionar de maneiras estranhas ou irá alterar o comportamento de casos limítrofes incomuns e introduzirá bugs muito sutis no programa. Você não quer fazer isso, certo?

O software deve ser “soft”, e não “hard”. Apesar disso, o medo nos leva a congelar o nosso código em uma tentativa de evitar que ele pare de funcionar. É a rigidez mortal do software.

> **PONTO-CHAVE** Não deixe seu código embalsamado. Se você tiver um código “inalterável” em seu produto, ele irá apodrecer.

Vemos a rigidez mortal surgir quando os autores originais saem de um projeto e ninguém que permaneça compreende totalmente o seu velho código, crucial aos negócios. Quando for difícil trabalhar com um código legado ou até mesmo fazer uma estimativa confiável para trabalhar com ele, os programadores devem evitar o núcleo do código. Ele vai se tornar uma selva de código indomável, em que feras digitais selvagens vagarão livremente. Para trabalhar de maneira previsível e no prazo, novas funcionalidades são adicionadas como módulos satélites em torno das fronteiras.

Vemos a rigidez mortal surgir quando um produto é instalado nos servidores de produção e é usado por muitos clientes diariamente. As APIs do sistema original são mantidas porque custará muito caro alterá-las, já que muitas equipes e muitos serviços agora dependem delas.

O código *jamais* deve permanecer imutável. Nenhum código é sagrado. Nenhum código jamais será perfeito. Como poderia ser? O mundo está constantemente mudando ao redor dele. Os requisitos estão sempre em transformação, independentemente do zelo com que tenham sido capturados. A versão 2.4 do produto é tão radicalmente diferente da versão 1.6 que é totalmente possível que a estrutura interna do código *deva* ser totalmente diferente. E estamos sempre encontrando novos bugs em nosso código antigo que devem ser corrigidos.

Se o seu código se tornar uma camisa de força, você estará lutando contra o software, e *não* desenvolvendo-o. Você estará permanentemente dançando em volta de uma lógica necrosada e criando caminhos mais tortuosos ainda em torno de um design não confiável.

> **PONTO-CHAVE** Você é o mestre de seu software; ele está sob seu controle. Não deixe que o código ou os processos em torno dele determinem a maneira como o código cresce.

Mudanças sem medo

É claro que é perfeitamente sensato ter medo de introduzir falhas no código. Projetos de software de grande porte contêm inúmeras sutilezas e complexidades que devem ser conhecidas. Não queremos introduzir bugs por meio de uma modificação descuidada. Somente os tolos farão alterações despreocupadamente, sem realmente saber o que estão fazendo. Isso seria uma “codificação cowboy”.

Então como reconciliar uma modificação corajosa com o medo de errar?

- Aprenda a fazer boas alterações – há práticas que aumentam a segurança de seu trabalho e reduzem as chances de erro. A coragem resulta de uma confiança de que sua modificação será segura.
- Saiba o que faz um software ser fácil de ser modificado e se esforce para criar softwares com esses atributos.
- Faça melhorias diárias em seu código para que ele se torne mais maleável. Recuse-se a comprometer a qualidade do código.
- Adote atitudes saudáveis que resultem em um código próspero.

Porém, acima de tudo, *simplesmente faça a alteração!* Sem medo. Você pode falhar; a alteração pode dar errado. Mas é sempre possível restaurar o código retornando-o a um estado funcional e tentar novamente. Você não deve ter vergonha de tentar e sempre aprenderá com seus erros. Basta certificar-se de que qualquer alteração feita tenha o apoio de testes e de inspeções suficientes antes de ser incluída em ambiente de produção.

Nada está gravado a ferro e fogo O design não está. A equipe não está. O processo não está. O código não está. Entenda isso e conheça a parte que cabe a você na melhoria de seu software.

> **PONTO-CHAVE** Para modificar um código, é preciso ter coragem e habilidade. Não seja descuidado.

Mude a sua atitude

Para “permitir” alterações saudáveis em seu código, a equipe de

programação deve adotar as atitudes corretas. Ela deve estar comprometida com a qualidade do código e deve *querer* realmente escrever um bom código.

Abordagens de codificação temerosas e covardes não são boas. Nós nos esquivamos: *não escrevi isso; parece uma porcaria; não quero ter nada a ver com isso; vou me envolver o mínimo possível com esse código*. Essas atitudes tornam a vida do programador um pouco mais fácil agora, porém resultam em um design ruim. Códigos antigos ficam estagnados ao mesmo tempo que novos entulhos se acumulam ao seu redor.

> **PONTO-CHAVE** Um "bom código" não é problema dos outros. É sua responsabilidade. Você tem autonomia para fazer uma alteração e realizar uma melhoria.

A seguir estão listadas as atitudes importantes tanto para a equipe quanto para os indivíduos que contribuem para um crescimento saudável do código:

- Corrigir um código incorreto, perigoso, ruim, duplicado ou desagradável não é uma distração, um desvio ou uma perda de tempo precioso. Deve ser positivamente incentivado. Com efeito, é esperado. Você não deve deixar pontos fracos contaminando o código por muito tempo. Se você vir códigos que sejam assustadores para modificar, então ele *deverá* ser alterado!
- A refatoração deve ser incentivada. Se você tiver uma tarefa que exija uma mudança fundamental no código, faça isso adequadamente: refatore. A equipe entende que isso é necessário e que algumas tarefas podem exigir um pouco mais de tempo quando nos deparamos com problemas como esse.
- Ninguém é "dono" de qualquer parte do código. Qualquer pessoa tem permissão para fazer alterações em qualquer seção. Evite o feudalismo no código; isso reduz a taxa de alterações.
- Cometer um erro ou escrever um código incorreto (pelo menos acidentalmente) não é nenhum crime. Se alguém corrigir ou melhorar o seu código, não será um sinal de fraqueza seu ou de que os outros programadores sejam melhores que você. Provavelmente, você fará

ajustes no trabalho deles amanhã. É simplesmente assim que funciona. Aprenda e cresça.

- A opinião de uma pessoa não deve ser considerada mais importante do que a de outra. Todos têm uma contribuição válida a fazer em qualquer parte da base de código. É claro que algumas pessoas têm mais experiência em determinadas áreas. Porém elas não são "donas" nem guardiãs do código sagrado. Tratar o trabalho de algumas pessoas como "mais preciso" ou "melhor" do que de outras coloca essas pessoas em um falso pedestal e rebaixa a contribuição do restante da equipe.
- Bons programadores *esperam* mudanças porque é disso que se trata o desenvolvimento de software. Você deve ter nervos de aço e não se importar com as mudanças no terreno aos seus pés. O código muda rapidamente; acostume-se com isso.
- Contamos com a rede de proteção da responsabilidade. Novamente, vemos as revisões, a programação aos pares e os testes (tanto os testes de unidade e de integração automatizados quanto as boas interações entre QA/desenvolvedores) como partes fundamentais para garantir que nosso código permaneça flexível. Se você fizer algo errado ou introduzir rigidez ao código, isso será identificado antes de se tornar um problema.

Faça a mudança

Uma história apócrifa conta que um turista, perdido em um vilarejo rural, parou um habitante local e pediu informações sobre o caminho para uma cidade em uma região distante. O habitante do vilarejo ficou alguns instantes pensativo e respondeu lentamente: *se estivesse indo para lá, eu não teria começado por aqui!*

Parece tolo, mas, com frequência, o melhor local para iniciar a sua jornada não é o lugar em que você está, em um lamaçal de código. Se tentar seguir em frente, você poderá afundar. Em vez disso, talvez seja melhor retornar a um local mais sólido, levar o seu código até uma rodovia local e, depois que estiver lá, seguir para o seu destino mais velozmente.

É óbvio que é importante aprender a percorrer um caminho pelo código – como mapeá-lo, rastreá-lo e entender em que locais ele oculta efeitos colaterais surpreendentes.

Crie o design visando a mudanças

Nós nos esforçamos para criar um código que incentive as alterações. É um código que revela o seu formato e o seu propósito e estimula a modificação por meio de simplicidade, clareza e consistência. Evitamos um código com efeitos colaterais porque ele será frágil diante de mudanças. Se você encontrar uma função que faça duas tarefas, separe-a em duas partes. Torne explícito o que estiver implícito. Devemos evitar um acoplamento rígido e uma complexidade desnecessária.

Quando uma base de código feia e rígida resiste a mudanças, precisamos de uma estratégia de batalha; melhoramos o código lentamente, dia após dia, efetuando melhorias seguras por partes. Fazemos alterações em linhas de código e na estrutura em geral. Durante um período de tempo, nós o observaremos passar gradualmente para um formato mais maleável.

PONTO-CHAVE Normalmente, é melhor fazer uma série de ajustes frequentes, pequenos e testáveis em vez de fazer uma alteração abrangente no código.

Não tente lutar com toda a base de código de uma só vez. Essa pode ser uma tarefa assustadora e, talvez, intratável. Em vez disso, identifique uma seção de código limitada com a qual você deva interagir e concentre-se em alterá-la.

Ferramentas para fazer alterações

Preste atenção nisto! É realmente importante: boas ferramentas podem ajudar a fazer alterações seguras de modo extremamente rápido.

Uma boa suíte automatizada de testes permite que você trabalhe bem e rapidamente. Ela permite fazer modificações e obter feedback rápido e confiável para saber se suas modificações fizeram com que algo deixasse de funcionar. Considere introduzir algum tipo de teste verificável para as seções de código escolhidas a fim de evitar erros. Assim como o código se beneficia com a responsabilidade e com processos cuidadosos de revisão,

o mesmo vale para os testes.

> **PONTO-CHAVE** Os testes automatizados representam uma estrutura de segurança de valor inestimável, que aumenta a confiança em suas alterações de código.

A espinha dorsal de seu desenvolvimento deve ser a *integração contínua*: um servidor que faz continuamente o checkout e gera a versão mais recente do código. Se – que os céus não permitam – algo ruim passar de modo que haja um erro no build, você descobrirá rapidamente. Os testes automatizados devem ser executados também no servidor de build.

Escolha suas batalhas

Nada está gravado a ferro e fogo, mas nem tudo deve ser fluido.

Naturalmente, devemos selecionar nossas batalhas. Não podemos alterar todo o código sempre, ao mesmo tempo que adicionamos mais trabalhos novos. Sempre encontraremos códigos desagradáveis que não poderemos corrigir no momento, independentemente do quanto gostaríamos de fazer isso. O trabalho poderá ser muito grande. Ou poderá estar além do escopo de uma refatoração gigantesca.

Há uma determinada quantidade de *débitos técnicos* com os quais devemos conviver até termos a oportunidade de efetuar melhorias no futuro. Isso deve ser incluído no plano do projeto. Um débito significativo transforma-se em um item a ser trabalhado e inserido no cronograma de desenvolvimento, em vez de ser esquecido, deixando que continue a contaminar o código.

Mais alterações

Parece um pesadelo. Quem possivelmente consegue trabalhar com um código que está constantemente sendo alterado? Já é difícil acompanhar diversas alterações simultâneas, quanto mais envolver-se com elas!

Entretanto devemos encarar o fato de que o código muda: qualquer código que permaneça inalterado será um peso. Nenhum código está livre de modificações. Evitar uma seção de código por medo de alterá-la é contraproducente.

Perguntas

1. Quais atributos em particular tornam o software mais fácil de ser alterado? Você escreve um software desse tipo naturalmente?
2. Como podemos balancear a "ausência de um dono do código" com o fato de algumas pessoas terem mais experiência do que outras? Como isso afeta a alocação das tarefas entre os programadores?
3. Todo projeto tem códigos que mudam com frequência e códigos que mudam menos. O código desse último tipo pode ser menos alterado por não estar sendo usado, por ter um design saudável e receber extensões por meio de módulos externos ou porque as pessoas o evitam ativamente por ser um código ruim. Quanto de cada um desses tipos de código rígidos você tem?
4. As ferramentas de seu projeto apoiam suas alterações de código? Como isso pode ser melhorado?

Veja também

- *Jogando segundo as regras* (Capítulo 15) – "Nada está gravado a ferro e fogo" é uma das três regras complementares criadas pela minha equipe.
- *O fantasma de um código do passado* (Capítulo 5) – Espere alterar o código regularmente e aprenda com cada mudança feita.
- *Estará pronto quando estiver pronto* (Capítulo 32) – Nenhum software jamais estará "completo". Ele é "soft" e poderá mudar de várias maneiras no futuro. Entretanto é importante saber quando o trabalho atual no software está terminado.
- *Chafurdando na lama* (Capítulo 7) – Descreve algumas técnicas para fazer alterações corajosas.
- *O curioso caso do código congelado* (Capítulo 22) – Um código congelado é o oposto de um código dinâmico e mutável.

TENTE ISTO... Identifique um código em seu projeto no qual ninguém queira mexer. É adequado retrabalhar esse código agora? Descubra como melhorá-lo.

10.000 MACACOS

(OU ALGO POR AÍ)

COMO EVITAR QUE SEU SOFTWARE FIQUE

CRAVADO EM PEDRA

A OTIMIZAÇÃO TEM SIDO DESCRITA COMO UMA ESPADA DE DOIS GUMES

"ESPADA DE DOIS GUMES": ALGO QUE TEM TANTO UMA VANTAGEM QUANTO UM CUSTO OU UM RISCO OCULTO

ESSE É UM FATO QUE SERVE DE AVISO SOBRE COMO ESPADAS FAMOSAS FORAM CRAVADAS EM PEDRA

EXCALIBUR

PORTANTO ESCREVA UM CÓDIGO LENTO E NÃO OTIMIZADO PARA EVITAR QUE ELE SEJA CRAVADO EM PEDRA

CAPÍTULO 19

Um estudo sobre reutilização de código

Se algo puder ser reduzido, reutilizado, consertado, reconstruído, renovado, reacabado, revendido, reciclado ou reaproveitado, ele deverá ser restringido, reprojetado ou removido de produção.

– Pete Seeger

Ouvimos falar de algo mítico chamado "reutilização de código". Durante um tempo, isso estava incrivelmente na moda; outra bala de prata do software – uma novidade para os fornecedores espertos venderem. Eu não compro essa ideia.

Com frequência, falamos em termos de "casos de uso" quando desenvolvemos software. Também vemos os *casos de reutilização* a seguir:

Caso de reutilização 1: copiar e colar

É o código copiado de uma aplicação e transplantado cirurgicamente para outra. Bom, em meu livro, isso é mais *duplicação de código* do que *reutilização*. Ou, de modo menos educado, uma *programação copy-and-paste* (copiar e colar). Em geral, ela é perversa; é o equivalente à *pirataria de código*. Pense em um bando de programadores fanfarrões pilhando e acumulando joias de software de bases de código ricas pelos sete mares de software. Ousado. Porém perigoso. É uma codificação com os péssimos hábitos de higiene de um marujo.

Lembre-se do mantra DRY: *Do not Repeat Yourself* (Não se repita).

Esse tipo de "reutilização" é um *verdadeiro* assassino quando o mesmo fragmento de código é repetido 516 vezes em um projeto e você descobre que ele contém um bug. Como você garantirá que irá encontrar e corrigir todas as manifestações do problema? Boa sorte com isso.

Apesar do que foi dito, você pode argumentar que copiar e colar código entre projetos faz com que um trabalho seja feito. Há muito disso por aí e o mundo não acabou. Ainda. E um código copiado e colado evita o acoplamento desnecessário de que um código excessivamente DRY pode sofrer.

Entretanto copiar e colar é um negócio ruim, e nenhum programador que se preze admitirá esse tipo de "reutilização" de código.

> **PONTO-CHAVE** Evite codificar copiando e colando. Fatore sua lógica em funções compartilhadas e bibliotecas comuns, em vez de sofrer com códigos (e bugs) duplicados.

Embora seja tentador copiar e colar código entre arquivos em uma base de código, é mais tentador ainda copiar seções grandes de código da Internet. Todos nós já fizemos isso. Você pesquisa algo online (sim, o Google é uma ótima ferramenta para os programadores, e bons programadores sabem manejá-lo muito bem). Você encontra um trecho de código de aparência bem plausível em um fórum ou em uma postagem de blog. E você o insere diretamente em seu projeto para ver se funciona. *Ah, isso parece que serve.* Faça commit.

Embora seja incrível que almas gentis disponibilizem tutoriais e exemplos de código online para nos esclarecer, é perigoso usar isso diretamente, sem fazer uma avaliação crítica antes de incorporar esses recursos em seu trabalho.

Considere os seguintes aspectos previamente:

- O código está *totalmente* correto? Ele trata todos os erros adequadamente ou é somente ilustrativo? (Com frequência, deixamos o tratamento de erros e os casos especiais como um *exercício para o leitor* quando publicamos exemplos.) O código está livre de bugs?
- É a melhor maneira de conseguir o que precisamos? É um exemplo desatualizado? Ele vem de uma postagem de blog realmente antiga, que contém código anacrônico?
- Você tem o direito de incluir esse trabalho em seu código? Há algum termo de licença que seja aplicável?

- Com que abrangência você o testou?

PONTO-CHAVE Não copie códigos encontrados na Web em seu projeto sem inspecioná-lo cuidadosamente antes.

Caso de reutilização 2: faça o design visando à reutilização

O design de uma biblioteca é feito desde o princípio para que ela seja incluída em vários projetos. *Evidentemente*, isso é mais teologicamente correto do que fazer uma programação horrível, em que o código seja copiado e colado. Entretanto me desculpe: isso não é “reutilizar” código. É *usar* código. A biblioteca foi projetada para ser usada dessa maneira desde o início!

Essa abordagem também pode representar um trabalho adicional enorme e desnecessário.

Mesmo que você suspeite que uma seção vá ser usada em mais de um projeto, em geral, não vale a pena projetá-la visando a usos múltiplos desde o início. Fazer isso pode resultar em um software excessivamente complexo e inchado, com APIs muito complexas que tentem atender a todos os casos de uso em geral. Em vez disso, empregue o princípio *YAGNI* (You Aren’t Going to Need It, ou Você não irá precisar disso): se você *não vai precisar de algo* (ainda), não implemente (ainda).

Foque na implementação do código mais simples possível que satisfaça os requisitos no momento. Escreva somente o que for necessário e crie a menor e mais apropriada API possível.

Em seguida, quando outro programa quiser incorporar esse componente, você poderá adicionar e estender o código existente em funcionamento. Ao produzir somente a menor quantidade de software possível, o risco de introduzir bugs ou de criar APIs desnecessárias às quais você deverá dar manutenção durante muitos anos será reduzido.

Com frequência, o seu segundo “uso” planejado jamais se materializará, ou o segundo usuário terá requisitos surpreendentemente diferentes daqueles esperados por qualquer pessoa.

Caso de reutilização 3: promover e refatorar

Escreva seções de código pequenas e modulares. Mantenha-as limpas e organizadas.

Assim que você perceber que o código deve ser usado em mais de um local, refatore-o: crie uma biblioteca ou um arquivo de código compartilhado. Transfira o código para lá. Estenda a API o *mínimo possível* para acomodar o segundo usuário.

Nessa fase, é tentador pensar que a interface deve ser limpa, retrabalhada e complementada. Mas essa, definitivamente, pode não ser uma boa ideia. Procure manter as alterações mínimas e simples, pois:

- Seu código existente funciona. (Ele funciona bem, certo? E você tem os testes para provar?!) Toda alteração gratuita que você fizer distancia o código desse estado de funcionamento.
- É possível que um terceiro cliente apareça logo depois, com requisitos levemente diferentes. Seria uma pena (assim como um desperdício de esforço) ter de alterar novamente a API ajustada e adaptá-la.

> **PONTO-CHAVE** O código deve ser “compartilhado” porque será útil a vários clientes, e não porque os desenvolvedores queiram criar uma biblioteca compartilhada elegante.

Caso de reutilização 4: compre ou reinvente a roda

Quando for necessário acrescentar um novo recurso, talvez já haja bibliotecas de terceiros disponíveis que ofereçam a funcionalidade.

Considere cuidadosamente se faz mais sentido do ponto de vista econômico desenvolver o seu próprio código, inserir uma versão de código aberto (se os termos da licença permitirem) ou comprar uma solução de terceiros com suporte do fornecedor.

Você deve avaliar os custos de comprar algo pronto em relação aos custos de desenvolvimento, a provável qualidade do código e a facilidade de integração e de manutenção de cada solução. Os desenvolvedores tendem a querer escrever tudo por conta própria não só pelo exercício intelectual,

mas também por uma falta de confiança no desconhecido. Tome uma decisão bem fundamentada.

> **PONTO-CHAVE** Não despreze o código de outras pessoas. Pode ser melhor usar bibliotecas existentes em vez de escrever a sua própria versão.

Perguntas

1. Que volume de duplicação há em sua base de código? Com que frequência você vê um código copiado e colado entre funções?
2. Como você pode determinar o nível de diferença que deve haver entre seções de código para que seja aceitável considerar que não *haja* duplicação e que você não deve tentar refatorar as versões?
3. Você copia exemplos de código de livros ou de sites e os insere em seu trabalho com frequência? Qual é o nível de esforço que você investe para "sanitizar" o código para que ele seja incluído? Você atualiza o layout, os nomes das variáveis etc. sem pensar duas vezes? Você adiciona testes?
4. Ao adicionar códigos da Internet, você deve incluir comentários em torno dele indicando a fonte da implementação? Por quê?

Veja também

- *Mantenha a simplicidade* (Capítulo 16) – Uma reutilização adequada mantém a simplicidade de seu código e evita os tipos de problema que vimos no capítulo 7 (Chafurdando na lama).
- *Percorrendo um caminho* (Capítulo 6) – Uma duplicação desnecessária dificulta a navegação por uma base de código.

> **TENTE ISTO...** Se você estiver trabalhando em qualquer código desnecessariamente genérico, descubra uma maneira de eliminar essa generalidade e manter somente a interface essencial da lógica útil.

10.000 MACACOS
(OU ALGO POR AÍ)
VOCÊ TEVE DE TRABALHAR MUITO E ARDUAMENTE PARA COMPÔ-LO. POR QUE OUTRAS PESSOAS DEVERIAM TER SEU CÓDIGO DE GRAÇA?
1. OCULTE O CÓDIGO
namespace myCode
{
void doGoodStuff();
void greatThings();
void wellNamed();
void foo();
}
SEMPRE ESCOLHA O NOME MENOS ÓBVIO COLOQUE TUDO AQUI

2. OFUSQUE O CÓDIGO
COMPÊNDIO COMPLETO DE CÓDIGOS ENIGMÁTICOS
PARA GANHAR MAIS PONTOS, COLOQUE A CULPA EM SEU PADRÃO DE CODIFICAÇÃO

COMO EVITAR ESCREVER UM CÓDIGO REUTILIZÁVEL
OU, O MELHOR DE TUDO...
3. NÃO ESCREVA O CÓDIGO!
?
ONDE ESTÁ TODO O CÓDIGO?!!!
O CÓDIGO MENOS "REUTILIZÁVEL" É AQUELE QUE AINDA NÃO EXISTE.
ELE AINDA NEM FOI USADO, QUANTO MAIS REUTILIZADO.

10.000 MACACOS
(OU ALGO POR AÍ)

CAPÍTULO 20

Controle eficiente de versões

Tudo muda, nada desaparece.

– Ovídio

Para o desenvolvedor, controlar as versões é como comer e respirar; é como ter um editor de código-fonte e um compilador. É uma parte essencial da vida cotidiana de desenvolvimento.

Controle de versões é o processo de administrar várias revisões de um conjunto de arquivos. Esses, comumente, são os arquivos-fontes de um sistema de software (portanto, com frequência, o processo é chamado de *controle de fontes*), mas poderiam simplesmente ser revisões de uma árvore de documento ou de qualquer item que você armazenaria em um sistema de arquivos.

É um recurso bem simples. Porém um bom sistema de controle de versões bem usado apresenta uma série de vantagens:

- Oferece um ponto central de cooperação, coordenando o modo como os desenvolvedores trabalham juntos.
- Define e divulga o estado da arte; nenhum código será integrado, a menos que esteja armazenado no sistema. Outras ferramentas se ligam a esse sistema atualizado – por exemplo, integração contínua, sistemas responsáveis pela disponibilização de versões e sistemas de auditoria de código.
- Mantém um histórico do trabalho em um projeto, arquivando o conteúdo exato incluído em cada versão específica disponibilizada. É a máquina do tempo do código.

Esse sistema facilita a *arqueologia de software*, armazenando as alterações feitas nos arquivos de modo que seja possível descobrir

quais delas compõem uma determinada funcionalidade. Ele cataloga quem alterou cada arquivo e por quê.

- Gera um backup centralizado de seu trabalho.
- Disponibiliza uma rede de proteção ao desenvolvedor. Deixa espaço para realizar experimentos, efetuar tentativas de modificação e restaura o estado anterior caso essas alterações não funcionem.
- Promove um ritmo e uma cadência no trabalho: você realiza uma parte do trabalho, testa e, em seguida, faz o check-in. Então você vai para a próxima parte do trabalho.
- Permite várias linhas de desenvolvimento concorrentes na mesma base de código, sem que haja interferências.
- Possibilita reversões: qualquer mudança na história do projeto pode ser identificada e revertida se descobrirmos que houve um erro.

Use-o ou você se arrependerá

Essa é uma lista impressionante. O controle de versões é a espinha dorsal de seu processo de desenvolvimento; sem ele, você não terá um suporte estrutural.

Portanto a primeira regra de ouro do sistema de controle de versões é: *use-o*. Desde o início de qualquer projeto, empregue um sistema de controle de versões. Sem mas nem porém.

A maioria dos *SCVs* (Sistemas de Controle de Versões) praticamente não exige nenhum esforço de instalação, portanto não há nenhuma verdadeira desculpa para deixar de usar um sistema como esse.[1] Até mesmo os protótipos mais simples (aqueles códigos perniciosos que, com muita frequência, crescem e se transformam em sistemas de produção) podem começar com seu próprio repositório e ter o histórico registrado.

> **PONTO-CHAVE** Use um sistema de controle de versões. Não é uma ferramenta opcional nem interessante de ter. É a espinha dorsal do desenvolvimento. Seu trabalho estará correndo riscos sem ele.

O software não é inerentemente seguro; códigos-fontes em um disco são como fumaça digital: podem desaparecer facilmente com um movimento

de mãos. Já perdi a conta da quantidade de vezes ao longo dos anos que apaguei algo errado ou fiz uma alteração incorreta e não tinha um ponto de referência ao qual retornar. Um sistema de controle de versões atenua esses problemas. Um SCV decente e leve estimula check-ins pequenos e frequentes, proporcionando um nível sério de isolamento contra a sua própria estupidez.

História de guerra: perda de dados distribuídos

De pé, do lado de fora de um restaurante, perguntei aos membros de uma equipe com a qual estava colaborando em que local eu poderia obter seus códigos-fontes. Implicitamente, eu queria dizer "qual servidor hospeda o repositório em que devo fazer checkout?". Essas pessoas trabalhavam em suas casas e estavam distribuídas pela cidade.

Elas pensaram por um tempo, olhando de uma para outra inquisitivamente. Então Dave disse: *Bill, está em seu computador, certo?* Surpreso com a resposta, investiguei mais a fundo.

O fato é que esses programadores não eram fãs de um sistema de controle de versões – era "trabalhoso demais". Eles achavam que um SCV era pesado e orientado a processos. Eles preferiam se alternar para "tomar conta" do código. Todas as alterações eram enviadas por email no final de cada semana para que o proprietário as reunisse em um conjunto enorme que era enviado de volta novamente a todos.

Sim, com frequência, havia vários conflitos de código a serem solucionados; eram tratados por meio de palpites, nem sempre bem-sucedidos. Sim, havia perdas ou esquecimentos de vez em quando. Sim, não havia nenhum backup. E sim, o código-fonte foi catastroficamente perdido algumas vezes ao longo dos anos.

Mas, mesmo assim, eles estavam convencidos de que um sistema de controle de versões era muito formal – um trabalho excessivo. Sim, eles estavam satisfeitos em trabalhar dessa maneira.

Não foi há tanto tempo assim. Desde então, tenho evitado a equipe.

Escolha um, qualquer um

Tem havido vários sistemas diferentes de controle de versões produzidos ao longo dos anos, desde o comando pioneiro `rcs` do Unix (que data do início dos anos 80), passando pelos CVS centralizados (populares nos anos 90), o Subversion – seu primo moderno (que reinou nos anos 2000) – e o moderno mundo dos sistemas distribuídos como o Git e o Mercurial (que agora dominam os anos 2010). Algumas ferramentas são comerciais e muitas têm código aberto. Elas diferem quanto à licença, ao custo, à facilidade de uso, ao suporte a plataformas, à maturidade, à escalabilidade e ao conjunto de recursos.

Uma diferença fundamental está no modo de operação. Os sistemas *centralizados* históricos fazem toda a comunicação se afunilar para um servidor central que hospeda o *repositório* de todos os arquivos com versões controladas. É um modelo simples, porém exige que haja um acesso a esse servidor para qualquer operação que não seja trivial. Os SCVs mais recentes têm modelo *distribuído* – uma abordagem peer-to-peer (um a um), em que cada computador pode armazenar a sua própria cópia do repositório. Isso possibilita fluxos muitos mais impressionantes de trabalho, além de permitir que você interaja com o repositório mesmo não havendo uma conexão com a rede.

Qual ferramenta você usaria se pudesse escolher?

Dê prioridade a um sistema moderno, com suporte e que seja convencional. Até pouco tempo, provavelmente o Subversion era a opção-padrão devido ao seu custo (gratuito), à variedade de plataformas suportadas (praticamente todas, incluindo a sua torradeira) e à facilidade de uso. Entretanto, recentemente, o Git tem roubado essa coroa. Sistemas distribuídos de controle de versões têm se tornado mais populares por um bom motivo. Eles possibilitam fluxos de trabalho mais eficientes, verdadeiramente úteis. Contudo essa capacidade tem um custo: o Git, definitivamente, tem uma curva mais acentuada de aprendizado.[2]

Armazenando os itens corretos

Criamos tantos arquivos que eles ficam saindo pelas nossas orelhas. Temos arquivos-fontes, arquivos de configuração, dados binários, scripts

de build, arquivos intermediários de build, arquivos objetos, bytecodes, executáveis compilados etc. Quais deles devem ser armazenados no sistema de controle de versões?

Para nossos projetos com código-fonte, há duas respostas diferentes. Elas não são totalmente contraditórias.

Resposta um: armazene tudo

Você deve armazenar *todo* arquivo que seja necessário para recriar o seu software. Não importa se ele seja um arquivo "binário" ou um arquivo "fonte". Adicione-o ao sistema de controle de versões. Um bom SCV consegue lidar com binários grandes de modo razoável, portanto não é necessário se preocupar em administrar arquivos binários. (E se seus binários não fossem armazenados em um SCV, você ainda teria de arquivá-los e administrar suas versões em outro local, de qualquer modo.)

Começando com um computador de build devidamente configurado e com o sistema operacional e o ambiente corretos de compilação (as ferramentas de build, as bibliotecas-padrão etc., além de espaço suficiente em disco), uma simples operação de checkout deverá fornecer uma boa árvore de códigos-fontes para um build.

Isso significa que o seu repositório deve incluir:

- todos os arquivos-fontes de código;
- toda a documentação;
- todos os arquivos de build (makefiles, configurações de IDE, scripts);
- todos os arquivos de configuração;
- todos os recursos (imagens, sons, mídias a serem instaladas, arquivos de recursos);
- qualquer arquivo fornecido por terceiros (por exemplo, bibliotecas de código das quais você dependa ou DLLs de outras empresas).

Resposta dois: armazene o mínimo possível

É óbvio que você tem muito que armazenar. Porém não inclua itens desnecessários que irão confundir, inchar e atrapalhar. Mantenha a

estrutura de arquivos do repositório o mais simples possível. De modo específico:

- Não armazene arquivos de configuração de IDEs nem arquivos de cache. Evite fazer check-in de arquivos de header pré-compilados ou informações de código dinamicamente geradas, arquivos ctags, arquivos de configuração de preferências de usuários etc.
- Não armazene artefatos que tenham sido gerados – não é preciso efetuar check-in de arquivos objetos, arquivos de biblioteca ou binários de aplicações se eles forem um resultado do processo de build. Não é preciso nem mesmo fazer check-in de arquivos-fontes gerados automaticamente.

Às vezes, fazemos check-in de arquivos gerados automaticamente – caso eles sejam particularmente difíceis de serem gerados ou demorarem muito tempo para serem criados. Essa decisão deve ser tomada com muito cuidado – não polua o seu repositório com porcarias desnecessárias.

- Não armazene itens que não façam parte de seu projeto, como os instaladores de ferramentas de desenvolvimento ou a imagem de um sistema operacional para o servidor de build.
- Não faça check-in de relatórios de testes ou de bugs. Esses devem ser administrados por um sistema de relatório de bugs em outro local.
- Emails "interessantes" relacionados ao projeto não pertencem ao repositório. Se contiverem informações úteis, eles deverão ser colocados em arquivos de documentação mais estruturados.
- Não armazene configurações pessoais, por exemplo, o esquema de cores de seu editor, configuração de visões de seu IDE ou (particularmente) qualquer configuração que descreva a localização dos arquivos de build em seu computador.[3] Isso é especialmente ruim quando suas configurações entrarem em conflito com o computador de outro usuário.
- Não mantenha itens dos quais você *ache* que poderá precisar algum dia no repositório. Lembre-se de que você *pode* apagar itens que estão no sistema de controle de versões se eles não estiverem relacionados ao

estado da arte atual (é perfeitamente seguro – eles continuarão arquivados). Não se atenha a bagagens digitais que possam ser jogadas fora.

PONTO-CHAVE Armazene *todo* arquivo que componha o seu projeto de software em um sistema de controle de versões. Contudo armazene *o mínimo possível*; não inclua nenhum arquivo desnecessário.

Armazenando versões de software

Você deve colocar as versões de software disponibilizadas em seu sistema de controle de versões? Algumas empresas colocam todas as versões disponibilizadas em um repositório. Em geral, é um repositório separado de "versões"; os binários não devem ficar realmente juntos com os arquivos-fontes.

Considere arquivá-los em uma estrutura simples e estática de diretórios em outro local. O sistema de controle de versões não traz muitas vantagens por registrar estruturas menos dinâmicas de arquivos para a posteridade. Pode ser mais fácil navegar em um servidor de arquivos para esse tipo de arquivamento.

Layout do repositório

Pense cuidadosamente no layout de seu repositório. Garanta que a estrutura de diretórios seja clara e que revele o formato do código. Inclua um documento "read me" útil no nível mais alto.

Evite duplicações a todo custo. Assim como as duplicações em seu código resultam em bugs, o mesmo ocorre com a duplicação de arquivos em um repositório.

Administre os códigos de terceiros cuidadosamente. Mantenha-os separados de seu próprios arquivos-fontes. Coloque-os em subdiretórios claramente identificados. Isso ajuda a monitorar as alterações de terceiros sem que haja confusão com seus arquivos.

Garanta que o seu repositório esteja configurado para ignorar arquivos inadequados. Você pode instruir a maioria dos sistemas para que ignorem determinados arquivos de acordo com regras de correspondência com

padrões. Isso ajuda a evitar que você faça check-in acidentalmente de arquivos pessoais de configuração, arquivos derivados e itens semelhantes.

Utilize bem o sistema de controle de versões

Mudar e mudar para melhor são duas coisas diferentes.
– Provérbio

Se a regra de ouro é "use um sistema de controle de versões", então a regra de prata será "use *bem* um sistema de controle de versões". É importante entender realmente o funcionamento de seu sistema de controle de versões e conhecer as melhores práticas para trabalhar com ele.

Várias dessas práticas são universais.

Faça commits atômicos

Os commits das alterações feitos no repositório narram a história de seu trabalho com o código. Considere como você narra essa história para que o histórico registrado esteja claro.

Faça commits pequenos e atômicos. Eles são mais fáceis de entender e é mais fácil verificar se estão corretos. É a estratégia de check-ins *pequenos e frequentes*.

> **PONTO-CHAVE** Faça check-ins *pequenos e frequentes* de suas alterações.

Não acumule o trabalho de uma semana até fazer check-in. Ou até mesmo de um dia. Os problemas a seguir podem ocorrer:

- É mais difícil rastrear as alterações feitas no código, pois elas serão maiores e menos granulares.
- O restante do repositório de código poderá ter sido alterado massivamente entre suas atualizações. Seu novo trabalho poderá não ser mais válido.
- Se o mundo ao seu redor mudar, é mais provável que haverá conflitos: você pode ter alterado a mesma seção de código que outra pessoa e agora um conjunto comum de alterações deverá ser resolvido.

Commits atômicos são coesos e coerentes, apresentando alterações

relacionadas como um passo individual. Não faça um check-in que abranja mais de uma alteração. Se você se vir escrevendo uma mensagem de commit como *refatorando a estrutura interna e deixando o botão verde*, está claro que você realizou duas tarefas. Implemente-as em dois commits separados. Como um exemplo específico e comum, não altere o layout do código e uma funcionalidade ao mesmo tempo.

Um commit atômico é completo. Não faça check-in de trabalhos pela metade. Cada commit deve representar um passo completo.

Enviando as mensagens corretas

Em cada commit, forneça uma boa *mensagem de check-in*. Ela deve começar com um resumo *breve* do que mudou; o ideal é que isso seja feito em uma sentença clara. Então prossiga com os motivos pelos quais você fez a mudança se isso for do interesse de alguém. Se for apropriado, inclua um número de referência de bug ou outras informações complementares.

Deixe a mensagem clara, sucinta e evite ambiguidades, como deve ocorrer com um bom código. Lembre-se do princípio DRY: Don't Repeat Yourself (Não se repita). Não há nenhuma necessidade de listar os arquivos alterados; o SCV já registra isso para você.

Procure criar mensagens que sintetizem a mudança na primeira sentença. Isso possibilita uma boa pesquisa em uma lista de commits quando você navegar pelo histórico do repositório.

A seguir, temos alguns exemplos de mensagens reais de check-in retiradas de uma única base de código. Quais delas você acha que são boas e quais são ruins?

- correção #4507: janelas utilitárias carregam atrás de ACVS
- adicionando alguns créditos.. corrige um bug que fazia com que a aba de exemplo no modo de edição não funcionasse..
- “” (*… sim, uma string vazia; é uma mensagem de commit surpreendentemente comum*)
- um desvio foi corrigido
- Um código muito assustador executado na carga do programa foi

documentado enquanto uma falha foi corrigida. Sabe, às vezes, eu realmente me desespero com o que vejo nessa base de código.

- Falando sério, alguém lê isso?

Componha bons commits

Um programador zeloso é atencioso e faz check-ins apropriados. Assim como a mensagem de commit deve ser bem composta, o mesmo deve ocorrer com o conteúdo do commit.

- Não provoque falhas no build. Antes de fazer check-in em qualquer código, teste-o antes em relação à versão mais recente do repositório. Isso garantirá que o seu novo código não provocará falhas no build nem irritará os demais desenvolvedores. Outros componentes dos quais o seu código dependa poderão ter sido alterados desde que você o escreveu, fazendo com que o seu novo trabalho esteja incorreto.

O processo mais simples é: fazer uma alteração, verificar se o build pode ser feito em relação à versão mais recente do repositório, verificar se o build funciona e efetuar o check-in. Nessa ordem. Sempre. Quando você estiver correndo porta afora para pegar um ônibus, pode ser muito tentador fazer um check-in apressado de um código que "deveria funcionar". Acredite em mim: raramente funciona.

- Não jogue um arquivo na lixeira nem remova-o, a menos que você saiba que ninguém mais o está utilizando. Isso é particularmente importante em projetos de sistemas para diversas plataformas, em que há vários builds.
- Não deixe que os editores briguem por cause de caracteres de fim de linha; essa é outra armadilha fácil em que podemos cair em projetos para diferentes plataformas.

Branches: vendo o bosque no lugar das árvores

Os branches são um recurso fundamental e importante do SCV. Eles permitem "bifurcar" seus esforços de desenvolvimento e trabalhar em recursos diferentes simultaneamente sem que as linhas de

desenvolvimento interfiram umas com as outras. Depois que o código em cada branch estiver concluído, cada branch poderá ser mesclado de volta à linha principal para sincronizá-lo com o seu pai. Essa é uma ferramenta de desenvolvimento imensamente eficaz.

Os branches podem ser usados para trabalhos pessoais (como uma área para desenvolvimentos feitos por um indivíduo ou para a realização de experimentos arriscados), para ajudar na cooperação da equipe (definindo áreas de integração ou de testes) e para gerenciamento de disponibilização de versões.

Muitas tarefas comuns se tornam muito mais simples com os branches. Considere usá-los para:

- Encapsular revisões da árvore de códigos-fontes. Por exemplo, cada funcionalidade pode ser desenvolvida em seu próprio branch.
- Trabalho de desenvolvimento exploratório – códigos que você não tem certeza de que irão funcionar. Não corra o risco de causar erros na linha principal de desenvolvimento: faça ajustes em um branch e, em seguida, mescle com a linha principal se o experimento for um sucesso. Você pode criar vários branches para testar diferentes maneiras de implementar a mesma funcionalidade; mescle com a tentativa mais bem-sucedida (uma espécie de seleção natural de código).
- Alterações importantes que mudem grande parte da árvore de códigos-fontes e que irão demorar para serem concluídas, exigindo muitos testes de QA e muitos check-ins individuais para que estejam corretas. Fazer esse trabalho em um branch evita que outros desenvolvedores fiquem parados por dias infindáveis, com uma árvore de código com falhas.
- Correções individuais de bugs. Crie um branch para trabalhar em uma correção de bug, testar o seu trabalho e, em seguida, mesclar o branch depois que a falha tiver sido corrigida.
- Separar desenvolvimentos voláteis de linhas estáveis para disponibilização de versões. Por exemplo, usamos *branches de release* (release branches) para "congelar" o código que compõe uma versão

de software a ser disponibilizada. Os branches de release serão descritos no capítulo 23 (*Por favor, libere a versão*).

Os branches são um ótimo recurso de organização, simplesmente à espera para serem usados. Não tenha medo deles. Não polua sua linha principal de desenvolvimento com distrações desnecessárias que possam ser isoladas em um branch.

No entanto saiba que os branches nem sempre são a técnica mais adequada para desenvolvimentos concorrentes. Às vezes, é melhor evitar esforços múltiplos de desenvolvimento concorrente, praticamente invisíveis (com o consequente overhead de integrações periódicas), em favor de uma abordagem simples, baseada na *alternância de funcionalidades* (feature toggle) na linha principal do código de desenvolvimento.[4]

Um lar para o seu código

O repositório do sistema de controle de versões é o lar para o seu código. Tome cuidado para que ele não se torne um asilo. Ou um necrotério.

Depois de ver uma quantidade suficiente de projetos de grande porte, você observará que qualquer código-fonte de um projeto razoavelmente complexo tende a se acostumar com o SVC em que estiver armazenado.

Isso acontece à medida que o projeto e sua infraestrutura crescem. À medida que o código passa da infância para a adolescência, os scripts de build e as ferramentas para disponibilização de versão se tornam profundamente integrados ao repositório. Por exemplo, scripts automatizados para atualização de versões determinam o funcionamento do sistema de controle de versões. Algumas convenções de estrutura de arquivos são seguidas porque o SCV as obriga (por exemplo, a existência de diretórios vazios ou a possibilidade de criação de links simbólicos).

Essas características tendem a moldar a maneira de trabalhar com o código, para o bem ou para o mal.

PONTO-CHAVE O código-fonte habita o SCV em que ele estiver armazenado. Quanto mais maduro estiver um projeto, mais profundamente ele dependerá de

seu *habitat*.

Não temos a tendência de migrar projetos entre SCVs com muita frequência, pois valorizamos o histórico de revisões registrado pelo repositório. A migração é possível, porém normalmente é um processo confuso e que gera perdas. Esse é um dos principais motivos para escolher um SCV apropriado no início de um projeto.

Conclusão

Melhorar é mudar; ser perfeito é mudar com frequência.

– Sir Winston Churchill

Um sistema de controle de versões é uma das ferramentas básicas de desenvolvimento de software. Todo programador deve saber como manejar um bom SCV, da mesma maneira que você deve ter um bom conhecimento para trabalhar com um editor eficiente de código-fonte.

O sistema de controle de versões forma a espinha dorsal para que uma equipe possa cooperar. É essencial para o desenvolvimento de software, porém pode ser usado para diversas finalidades: por exemplo, para administrar árvores de documentos. Este livro foi escrito como um conjunto de arquivos AsciiDoc mantidos no Git. Isso permitiu que eu tivesse facilmente um backup de meu trabalho, movesse arquivos entre computadores sem dores de cabeça, rastreasse as alterações que fiz e compartilhasse o manuscrito com minha editora.

O sistema de controle de versões é útil até mesmo em cenários em que não há cooperação; as informações mais importantes se beneficiarão com o fato de estarem armazenadas em um repositório. Por padrão, eu crio um repositório para todo protótipo de projeto que começo, mesmo que seja um projeto pessoal de estimação. Eu administro muitos outros itens com o Git, por exemplo, as configurações pessoais salvas no diretório home de meu computador. Isso faz com que seja extremamente simples configurar um novo computador com minhas preferências pessoais apenas clonando esse repositório.

Perguntas

1. Os sistemas de controle de versões vêm com uma GUI e com ferramentas de linha de comando. Quais são os prós e os contras de cada uma delas? É importante saber usar as duas opções? Por quê?
2. Quais são os possíveis problemas que os SCVs distribuídos introduzem em relação ao modelo centralizado, mais simples? Como é possível evitar esses problemas?
3. Você está usando o sistema de controle de versões correto? Quais recursos seu sistema atual não tem, mas você já viu em outro SCV?
4. Usar um sistema de controle de versões significa que não é necessário fazer backup de um computador pessoal de desenvolvimento?
5. Qual destes é um sistema mais seguro para trabalhar de forma concorrente: *alternância de funcionalidades* (feature-toggles) ou *branches* concorrentes? Qual dessas opções envolve menos overhead de gerenciamento e de integração?
6. Você está prestes a efetuar um commit de suas alterações em um repositório e percebe que trabalhou em duas tarefas diferentes. Você deve parar e retrabalhar o conjunto de alterações ou deve simplesmente efetuar o commit do código porque ele já foi implementado? Por quê? Como diferentes ferramentas SVC ajudam nesse tipo de situação?

Veja também

- *Melhore o código removendo-o* (Capítulo 4) – Um sistema de controle de versões permite apagar um código de forma confiante. É sempre possível tê-lo de volta a partir dos arquivos guardados.
- *Por favor, libere a versão* (Capítulo 23) – O sistema de controle de versões é uma parte essencial de um bom fluxo de disponibilização e de implantação de versões. Esse capítulo descreve os *branches de release* com mais detalhes.
- *O curioso caso do código congelado* (Capítulo 22) – Os *branches de release* correspondem ao mecanismo do SCV usado para garantir um *congelamento de código* enquanto o trabalho continua na linha principal de desenvolvimento.

- *Percorrendo um caminho* (Capítulo 6) – O histórico de seu repositório pode conter informações valiosas que ajudarão você a navegar pela sua base de código e avaliar a sua qualidade.

> **TENTE ISTO...** Preste atenção na qualidade de seus commits. Eles são frequentes, atômicos, pequenos e coerentes? Trabalhe no sentido de fazer alterações melhores.

---

1 Por exemplo, um simples comando `git init` em um diretório define um repositório Git em um piscar de olhos.

2 Estou tão imerso atualmente no fluxo de trabalho distribuído do Git que eu consideraria o uso de um repositório do Subversion somente se pudesse usar o Git como o cliente “frontend” para a conexão. Todos com quem converso compartilham essa experiência.

3 Nenhum sistema de build deve depender de localizações fixas em um computador. É um design ruim para um sistema de build. Corrija-o!

4 Uma *alternância de funcionalidades* (feature toggle) é a técnica em que um arquivo de configuração que habilita ou desabilita funcionalidades seletivamente é usado em seu software. Pode ser um arquivo de configuração XML interpretado em tempo de execução ou um conjunto de flags de pré-processador usado em tempo de compilação.

CAPÍTULO 21

Marcando um gol

As lutas não perdurariam se somente um dos lados estivesse errado.

– François de La Rochefoucauld

Os filósofos e disseminadores de cabelos vistosos e melodiosos de meados do século XX – os Beatles – nos disseram: *all you need is love* (tudo que você precisa é de amor). Eles enfatizaram a questão: *love is all you need* (amor é tudo de que você precisa). Amor – é isso. Literalmente. Nada mais.

É incrível a duração de suas carreiras, considerando que eles não precisavam comer nem beber.

Em nossos relacionamentos profissionais com outros habitantes da fábrica de software, definitivamente, nós nos beneficiaríamos com mais desse sentimento. Um pouco mais de amor poderia nos levar a um código muito melhor! A programação no mundo real é um empreendimento interpessoal e, portanto, está inevitavelmente ligada a problemas de relacionamento, política e atritos gerados pelos nossos processos de desenvolvimento.

Trabalhamos bem de perto com várias pessoas. Às vezes, em cenários estressantes.

Não será saudável para nossos relacionamentos profissionais nem para a consequente qualidade de nosso software se nossas equipes não estiverem trabalhando juntas harmoniosamente. Porém muitas equipes sofrem com esses tipos de problema.

Como uma tribo de desenvolvedores, um de nossos relacionamentos mais importantes é com a equipe de QA; principalmente porque interagimos com ela de forma muito próxima, normalmente nos pontos mais

estressantes do processo de desenvolvimento. Na pressa de lançar o software antes de um prazo ter se esgotado, tentamos chutar a bola do software entre os goleiros responsáveis pelos testes.

Vamos dar uma olhada nesse relacionamento agora. Veremos por que ele é estressante e por que não deve sê-lo.

Para que serve a equipe de QA?

Para alguns, o que essa equipe faz é óbvio. Para outros, é um mistério. O departamento de "QA" (ou seja, de *Quality Assurance*, ou Controle de qualidade) existe para garantir que seu projeto lance um produto de software com qualidade suficiente. Ele constitui uma parte necessária e vital do processo de construção.

O que isso implica? A resposta mais óbvia e prática é que esse departamento deve testar tudo o que os desenvolvedores criarem de modo a garantir que:

- O que foi criado atenda à especificação e aos requisitos, ou seja, que todas as funcionalidades que deveriam ter sido implementadas o foram.
- O software funcione corretamente em todas as plataformas, ou seja, funcione em todos os sistemas operacionais, em todas as versões desses sistemas operacionais, em todas as plataformas de hardware e em todas as configurações suportadas (por exemplo, atenda aos requisitos mínimos de memória, às velocidades mínimas do processador, à largura de banda da rede etc.).
- Nenhuma falha tenha sido introduzida no build mais recente, ou seja, que as novas funcionalidades não gerem falhas em nenhum dos demais comportamentos e que nenhuma regressão (a reintrodução de um comportamento anterior ruim) tenha sido introduzida.

Seu nome é "QA", e não simplesmente "departamento de testes", e isso tem um motivo. Sua função não é somente apertar botões como se fossem robôs; é garantir a qualidade do produto.

Para isso, a equipe de QA deve estar profundamente envolvida em todo o processo de desenvolvimento, e não ser apenas um acréscimo no final.

- Eles participam da especificação do software para entender – e dar forma – ao que será criado.
- Contribuem com o design e a construção para garantir que o que será implementado será testável.

- Envolvem-se intensamente na fase de testes, naturalmente.
- E também na disponibilização final da versão: garantem que o que foi testado é o que será realmente disponibilizado e implantado.

Desenvolvimento de software: jogando adubo

Em locais de trabalho incultos, o processo de desenvolvimento é modelado como se fosse um enorme encanamento: materiais brutos são jogados na parte de cima, passam por diversos processos até se transformarem em jatos de software perfeitamente formados (bem, talvez gotas) no final. O processo ocorre de modo semelhante ao que está descrito a seguir:

1. Alguém (talvez um *analista de negócios* ou um *gerente de produto*) despeja alguns requisitos na boca do encanamento.
2. Esses passam pelos arquitetos e pelos designers, que os transformam em especificações e em diagramas bonitos (ou em boas intenções ou fumaça e espelhos).
3. Isso flui para os programadores (que fazem o *verdadeiro* trabalho, naturalmente) e se transforma em um código executável.
4. Em seguida, o fluxo segue para QA. Nesse ponto, um obstáculo é atingido: o software "perfeitamente formado", em um passe de mágica, se transforma em um desastre que não funciona. Essas pessoas *quebram* o código!
5. *Em algum momento*, os desenvolvedores empurram tudo cano abaixo para eliminar esse obstáculo e o software finalmente flui na extremidade final do encanamento.

Nos ambientes de desenvolvimento piores, esse encanamento lembra mais um esgoto. A equipe de QA tem a impressão de que os desenvolvedores estão empurrando esgoto para eles, em vez de lhe entregar um presente muito bem embrulhado, de forma bem planejada. As pessoas da equipe de QA acham que o software é despejado sobre elas em vez de estarem trabalhando em equipe.

O desenvolvimento de software é realmente tão linear assim? Nossos

processos realmente funcionam como esse simples encanamento (independentemente da pureza do conteúdo)?

Não. Não funcionam.

O encanamento é uma primeira aproximação interessante (afinal de contas, não é possível testar um código que ainda não tenha sido escrito), porém é um modelo simplista demais para um desenvolvimento de verdade. A visão de um encanamento linear é um corolário lógico da fascinação de longa data de nosso mercado pela metodologia problemática de desenvolvimento *em cascata* (warterfall).[1]

> **PONTO-CHAVE** Ver o desenvolvimento de software como um processo linear é errado.

Entretanto essa visão do processo de desenvolvimento explica por que a interação da equipe de desenvolvimento com a equipe de QA não é tão harmoniosa quanto deveria. Nossos processos e modelos de interação, com muita frequência, são moldados pela metáfora problemática do desenvolvimento baseado em esgoto. Devemos estar em constante comunicação, em vez de simplesmente despejar software sobre a equipe de QA no final do esforço de desenvolvimento.

Uma falsa dicotomia

As interações entre nossas equipes estão cheias de obstáculos porque elas são somente isto: interações entre equipes *separadas*. O pessoal de QA é considerado como uma tribo diferente, distinta dos desenvolvedores "importantes". Essa visão problemática e particionada da organização do desenvolvimento inevitavelmente resulta em problemas.

Quando QA e desenvolvimento são vistos como passos separados, como atividades separadas e, desse modo, como equipes separadas, uma rivalidade artificial e uma falta de conexão podem se desenvolver facilmente. Isso é fisicamente reforçado pela criação de áreas artificiais que separam os responsáveis pelos testes e os desenvolvedores. Por exemplo:

- As duas equipes têm gerentes diferentes e respondem a diferentes linhas de responsabilidade.

- As equipes não ficam no mesmo local, e suas escrivaninhas ficam em lugares bem diferentes (já vi a equipe de QA em conjuntos diferentes de mesas, em andares diferentes, em prédios diferentes e até mesmo – em um caso extremamente tolo – em outro continente).
- As estruturas das equipes, as políticas de recrutamento e a rotatividade esperada da equipe de funcionários são diferentes. Os desenvolvedores são recursos valorizados, enquanto os responsáveis pelos testes são vistos como mercenários baratos e substituíveis.
- E pior ainda: as equipes têm incentivos bem diferentes para concluir suas tarefas. Por exemplo, os desenvolvedores trabalham com a promessa de receber um bônus como pagamento caso concluam rapidamente uma tarefa, mas isso não ocorre com quem trabalha com testes. Nesse caso, os desenvolvedores se apressam em escrever o código (provavelmente de modo *ruim* porque estão com pressa). Eles então ficam muito irritados quando o pessoal de QA não sanciona seus códigos para que uma disponibilização de versão seja feita no prazo.

Nós reforçamos essa separação por meio de estereótipos: quem desenvolve *cria*, quem testa *quebra*.

Há uma dose de verdade nisso. Existem diferentes atividades e habilidades necessárias em ambos os campos. Porém elas não são atividades logicamente separadas e isoladas. Os responsáveis pelos testes não encontram falhas no software somente para que o sistema deixe de funcionar e os desenvolvedores fiquem em uma situação infernal. Eles o fazem para aperfeiçoar o produto final.

Eles estão lá para garantir a qualidade. É o Q em QA. E isso pode ser feito de modo eficiente somente em conjunto com os desenvolvedores.

> **PONTO-CHAVE** Tome cuidado para não promover uma separação artificial entre os esforços de desenvolvimento e de testes.

Ao separar essas atividades, geramos rivalidade e discórdia. Com frequência, os processos de desenvolvimento opõem o pessoal de QA como *vilões* contra os *heróis* desenvolvedores. A imagem dos responsáveis

pelos testes é de alguém de pé na porta, impedindo que uma corajosa versão de software saia. É como se eles estivessem identificando falhas *sem motivo* em nosso software. Eles implicam com minúcias.

É quase como se eles corressem pelas florestas, capturassem bugs selvagens e então os injetassem em nosso código perfeito.

Isso soa como algo tolo?

É claro que sim, quando você o lê aqui, mas é fácil começar a pensar desta maneira: *o código é bom; esse pessoal simplesmente não sabe usá-lo*. Ou: *eles trabalharam tanto tempo para encontrar um bug básico como esse; realmente, eles não sabem o que fazem*.

O desenvolvimento de software não é uma batalha. (Bem, não deveria ser.) Estamos todos do mesmo lado.

Corrija a equipe para corrigir o código

A famosa lei de Conway descreve de que modo a estrutura de uma organização – e, especificamente, as linhas de comunicação entre as equipes – determina a estrutura do software.[2] Ela é popularmente parafraseada como "se você tiver quatro equipes escrevendo um compilador, ele terá quatro passos". A experiência mostra que isso é bastante preciso. Da mesma maneira que a *estrutura* da equipe afeta o código, a *saúde* das interações faz o mesmo com o software.

> **PONTO-CHAVE** Interações que não sejam saudáveis entre as equipes resultam em um código não saudável.

Podemos melhorar a qualidade de nosso software e a probabilidade de gerar uma ótima versão ao cuidarmos desses problemas de saúde: devemos melhorar o relacionamento entre os desenvolvedores e a equipe de QA. Devemos trabalhar *juntos* em vez de declarar guerra. Lembre-se de que *amor é tudo de que você precisa*.

Esse é um princípio geral e se aplica de maneira muito mais abrangente do que apenas entre os desenvolvedores e a equipe de QA. Equipes com funcionalidades que se entrelaçam são muito úteis em todos os aspectos.

O princípio se reduz ao modo como interagimos e trabalhamos com a equipe de QA. Não devemos tratar os membros da equipe de QA como marionetes cujas cordas controlamos ou a quem jogamos um software ruim para que seja testado. Em vez disso, devemos tratá-los como colegas de trabalho. Os desenvolvedores *devem* ter uma boa afinidade com a equipe de QA: deve haver amizade e camaradagem.

Vamos dar uma olhada nas maneiras práticas de trabalharmos em conjunto com esses habitantes do reinado de QA. Faremos isso observando os principais pontos em que os desenvolvedores interagem com ela.

Mas nós temos os testes de unidade!

Somos programadores conscientes. Queremos criar um software bem sólido. Queremos compor ótimas linhas de código, com um design coerente que contribua com um produto incrível.

É isso que fazemos.

Sendo assim, empregamos práticas de desenvolvimento que garantam que o nosso código seja o melhor possível. Fazemos revisões, programamos aos pares e inspecionamos. E *testamos*. Escrevemos testes de unidade automatizados.

Temos testes! E eles foram bem-sucedidos! O software deve ser bom, certo?

Mesmo com testes de unidade saindo pelas nossas orelhas, ainda não podemos garantir que o nosso software seja perfeito. O código pode funcionar conforme a intenção dos desenvolvedores, com sinais verdes para os testes em todo o caminho. Porém pode ser que isso não reflita o que o software *deveria* fazer.

Os testes podem mostrar que todas as entradas previstas pelos desenvolvedores foram tratadas corretamente. Entretanto pode ser que isso não seja realmente o que o usuário irá fazer. Nem todos os casos de uso (e os *casos de abuso*) do software foram considerados previamente. É difícil considerar todos esses casos – software é algo incrivelmente complexo. O pessoal de QA é muito bom exatamente para pensar nesses casos.

Por causa deles, um rigoroso processo de testes e de QA continua sendo parte vital de um processo de desenvolvimento de software, mesmo que tenhamos testes de unidade abrangentes definidos. Os testes de unidade atuam como

nossas ações responsáveis para provar que o código é bom o suficiente antes de o entregarmos à equipe de testes.

Disponibilizando um build para a equipe de QA

Sabemos que o processo de desenvolvimento não é uma linha reta; não é um fluxo simples. Desenvolvemos iterativamente e disponibilizamos melhorias incrementais, seja na forma de um recurso novo ou de uma correção de bug que devem ser validados. É um ciclo que se repete diversas vezes. Durante o processo de construção, vamos gerar diversos builds que serão enviados para a equipe de QA.

Portanto precisamos de um processo harmonioso de build e de entrega da versão.

Isso é vital: a passagem de nosso código para a equipe de QA não deve ter falhas – o código deve ser gerado de forma responsável e deve ser cuidadosamente distribuído. Fazer menos que isso é um insulto aos nossos colegas de QA.

Devemos gerar o build com a atitude correta: fornecer algo para QA não é jogar um código desleixado, gerado em qualquer computador velho, por cima da cerca. Esse não deve ser um ato descuidado nem negligente.

Além do mais, lembre-se de que isso não é uma batalha: não temos como meta que uma versão se *esgueire* pela equipe de QA e evite habilmente a sua defesa. Nosso trabalho deve ser de alta qualidade e nossas correções devem estar corretas. Não disfarce os *sintomas* dos bugs mais óbvios e espere que eles não tenham tempo de perceber as perversidades subjacentes do software.

Em vez disso, devemos fazer tudo o que pudermos para garantir que forneceremos algo que valha o tempo e o esforço da equipe de QA. Devemos evitar qualquer erro tolo ou desvios frustrantes. Deixar de fazer isso é uma falta de respeito com eles.

PONTO-CHAVE Deixar de gerar um build para QA de forma planejada e cuidadosa mostra uma falta de respeito com a equipe de testes.

Isso significa que você deve seguir as diretrizes discutidas nas seções a seguir.

Teste o seu trabalho antes

Antes de gerar uma versão a ser disponibilizada, os desenvolvedores devem realizar o melhor trabalho possível para provar que o código está correto. Eles devem testar o trabalho feito com antecedência. Naturalmente, esse objetivo é mais facilmente alcançado com uma suíte abrangente de testes de unidade executada regularmente. Isso ajuda a identificar qualquer *regressão* no comportamento (ocorrências repetidas de erros anteriores). Os testes automatizados podem evitar erros tolos e embaraçosos que fariam a equipe de testes perder tempo e impediriam que problemas mais importantes fossem encontrados.

Com ou sem testes de unidade, os desenvolvedores *devem* testar a nova funcionalidade de modo a ficarem satisfeitos com o seu bom funcionamento, conforme necessário. Isso parece óbvio, mas, com muita frequência, alterações ou correções que deveriam "simplesmente funcionar" são disponibilizadas e provocam problemas embaraçosos. Ou um desenvolvedor vê o código funcionar em um caso simples, considera-o adequado para ser disponibilizado em uma versão e nem mesmo pensa na variedade de maneiras pelas quais o código poderia falhar ou ser utilizado indevidamente.

É claro que executar uma suíte de testes de unidade é apenas tão eficiente quanto a qualidade desses testes. Os desenvolvedores devem assumir a total responsabilidade por isso. O conjunto de testes deve ser completo e representativo. Sempre que uma falha for informada pela equipe de QA, testes de unidade que demonstrem o problema devem ser adicionados para garantir que essas falhas não reaparecerão após a correção.

Tenha um propósito ao disponibilizar uma versão

Quando uma nova versão estiver sendo disponibilizada para a equipe de QA, o desenvolvedor deve deixar claro *como* ele espera exatamente que essa versão funcione. Não gere uma versão e peça simplesmente para "*ver*

se essa funciona".

Descreva de forma clara e precisa quais funcionalidades novas foram e quais não foram implementadas: exatamente o que se sabe que funciona e o que não funciona. Sem essas informações, você não poderá determinar quais testes serão necessários. Você estará desperdiçando o tempo da equipe de testes. Isso deve ser comunicado nas *informações de release* (release notes).

É importante compor um conjunto bom e claro de *informações de release*. Coloque-as junto do build de maneira não ambígua (por exemplo, no arquivo de implantação ou com um nome de arquivo que coincida com o nome do instalador). O build deve receber um número (único) de versão (talvez com um *número de build* incremental para cada versão). As informações de release devem ter esse mesmo número de versão.

Para cada versão disponibilizada, as informações de release devem informar claramente o que foi alterado e quais áreas exigem mais esforços de teste.

Mais pressa, menos velocidade

Jamais apresse um build, independentemente de quanto isso possa ser tentador. A pressão para gerar um build é maior à medida que o prazo final se aproxima, porém também é tentador fazê-lo antes de deixar a empresa no final do dia. Apressar um trabalho como esse incentiva o uso de atalhos, a não conferir tudo de forma completa ou a não prestar muita atenção no que você está fazendo. É simplesmente fácil demais. E não é a maneira certa de disponibilizar uma versão para a equipe de QA. Não faça isso.

Se você se sentir como um estudante tentando desesperadamente fazer a lição de casa correndo para ter "alguma coisa" no prazo, sabendo que o professor ficará aborrecido e que ele fará você refazer a lição, então algo está errado! Pare. E pense.

> **PONTO-CHAVE** Jamais apresse a criação de um build. Você cometerá erros.

Alguns produtos têm requisitos mais complexos de testes do que outros.

Somente dispare uma execução cara de testes em diversas plataformas ou sistemas operacionais se você achar que vale a pena – quando uma quantidade de funcionalidades e de correções sobre as quais houve consenso tiver sido implementada.

Automatize

A automação de passos manuais sempre elimina o potencial para erros humanos. Portanto automatize o seu build ou o processo de disponibilização de versão o máximo possível. Se puder criar um único script que faça ckeckout automático do código, gere o build e execute todos os testes de unidade, crie os instaladores ou efetue a implantação em um servidor de testes e atualize o build com suas informações de release; você eliminará o potencial para a ocorrência de erros humanos em diversos passos. Evitar erros humanos com a automação ajuda a criar versões que sejam sempre devidamente instaladas e que não contenham regressões. O pessoal de QA irá amar você por isso.

Respeite

A entrega de código para a equipe de QA é o ato de gerar algo estável e passível de ser disponibilizado como versão, e não o ato de jogar o último build não testado para ela. Não jogue uma granada de código sobre a cerca nem bombeie um software com aspecto de esgoto para eles.

Ao receber um relatório de falha

Fornecemos um build ao pessoal de teste. É nosso melhor esforço até então, e temos orgulho dele. Eles trabalham com esse build. Então eles encontram falhas. Não aja como se estivesse surpreso. Você sabia que isso iria acontecer.

> **PONTO-CHAVE** Os testes somente irão revelar problemas que os desenvolvedores de software adicionaram ao sistema (por ação ou omissão). Se uma falha for encontrada, será a *sua falha!*

Ao encontrar um bug, a equipe de testes criará um *relatório de falha*: um relatório do problema que possa ser monitorado. Esse relatório pode ser

priorizado, administrado e, após ter sido corrigido, poderá ser posteriormente conferido para ver se não houve regressão.

É responsabilidade *da equipe de testes* fornecer relatórios de falha precisos e confiáveis e enviá-los de maneira estruturada e organizada – usando um bom sistema de rastreamento de bugs, por exemplo. No entanto as falhas podem ser mantidas em uma planilha ou até mesmo como histórias em um backlog de trabalho. (Já vi todas essas opções funcionarem.) Desde que um sistema claro, que registre e divulgue as mudanças de estado de um relatório de falhas, esteja definido.

Como devemos responder a um relatório de falhas?

Inicialmente, lembre-se de que a equipe de QA *não* existe para provar que você é um idiota e fazer com que você tenha uma péssima imagem. O relatório de falha não é uma afronta pessoal. Portanto não leve isso para o lado pessoal.

> **PONTO-CHAVE** Não leve os relatórios de falha para o lado pessoal. Eles não são um insulto pessoal!

Nossa resposta "profissional" deve ser "obrigado, vou dar uma olhada nisso". Simplesmente fique feliz porque foi a equipe de QA quem encontrou a falha, e não um cliente. Você *pode* se sentir desapontado pelo fato de um bug ter passado pela sua rede.

Você deverá ficar preocupado se estiver atolado em muitos relatórios de falha a ponto de não saber por onde começar – isso é um sinal de que algo muito básico está errado e que deve ser abordado. Se você estiver nesse tipo de situação, é fácil ficar ressentido com cada novo relatório que chegar.

É claro que não devemos mergulhar de cabeça em cada falha assim que ela for informada. A menos que seja um problema trivial, com uma correção supersimples, é quase certo que haverá problemas mais importante a serem resolvidos antes. Devemos trabalhar em cooperação com todas as pessoas-chaves do projeto (gerentes, especialistas em produto, clientes etc.) para chegar a um consenso sobre quais são os problemas mais prementes com os quais gastaremos o nosso tempo.

Pode ser que o relatório de falha esteja ambíguo, não esteja claro ou precise de mais informações. Se esse for o caso, trabalhe *com* quem o gerou para esclarecer os problemas de modo que ambos possam compreendê-lo totalmente, reproduzi-lo de forma confiável e saber quando ele foi resolvido.

A equipe de QA pode identificar bugs somente do desenvolvimento, mesmo que não seja uma falha cujo autor tenha sido *você* diretamente. Talvez ela tenha se originado de uma decisão de design sobre a qual você não tinha nenhum controle. Ou, quem sabe, ela estava oculta em uma seção de código que não foi escrita por você. Porém assumir a responsabilidade por *todo o produto*, e não apenas pela sua pequena parte da base de código, é uma atitude saudável e profissional.

Nossas diferenças nos tornam mais fortes

Sempre que você entrar em conflito com alguém, há um fator que pode fazer a diferença entre arruinar e aprofundar o seu relacionamento. Esse fator é a atitude.

– William James (filósofo e psicólogo)

Relacionamentos profissionais eficazes se originam das atitudes corretas dos desenvolvedores. Ao trabalharmos com os engenheiros de QA, devemos entender e explorar nossas diferenças:

- Os profissionais responsáveis pelos testes são bem diferentes dos desenvolvedores. Os desenvolvedores geralmente não têm a mentalidade correta para efetuar testes eficientes. Isso exige uma maneira particular de olhar para o software, além de habilidades e características particulares para que o trabalho seja bem feito. Devemos respeitar a equipe de QA por essas habilidades – elas são essenciais se quisermos ter um software de alta qualidade.
- Os responsáveis pelos testes estão inclinados a pensar mais como um usuário do que como um computador; eles podem fornecer feedbacks valiosos sobre a qualidade percebida no produto, e não somente sobre a sua correção. Ouça suas opiniões e valorize-as.
- Quando um desenvolvedor trabalha com uma funcionalidade, o

instinto natural é focar no *caminho feliz* – o modo como o código funciona quando tudo vai bem (quando todas as entradas forem válidas, o sistema estiver funcionando completamente com o máximo de CPU, não houver problemas de memória nem de espaço em disco e todas as chamadas de sistema funcionarem perfeitamente).

É fácil ignorar as diversas maneiras pelas quais o software pode ser usado incorretamente, ou deixar de considerar classes inteiras de entradas inválidas. Estamos programados para considerar o nosso código por meio dessas tendências cognitivas naturais. O pessoal de testes tende a não ficar preso a tais tendências.

- Jamais se deixe levar pela falácia de que a equipe de QA é composta somente de "desenvolvedores fracassados". Há uma concepção errônea e comum de que, de algum modo, esses profissionais sejam menos inteligentes ou menos capazes. É um ponto de vista problemático e deve ser evitado.

> **PONTO-CHAVE** Cultive um respeito saudável pela equipe de QA. Aprecie trabalhar com ela de modo a criar um software excelente.

Peças do quebra-cabeça

Devemos ver os testes *não* como a "última atividade" em um modelo de cascata clássico; o desenvolvimento simplesmente não funciona dessa maneira. Após ter atingido 90% do caminho em um processo de desenvolvimento em cascata e ter chegado aos testes, é provável que você vá descobrir que *outros* 90% de esforço são necessários para concluir o projeto. Não é possível prever quanto tempo será necessário para os testes, especialmente quando eles são iniciados tarde demais no processo.

Assim como o código se beneficia de uma abordagem em que os testes vêm antes, o mesmo vale para todo o processo de desenvolvimento. Trabalhe com o departamento de QA e obtenha suas opiniões bem cedo para que as especificações sejam testáveis, garantir que o expertise desses profissionais seja usado no design do produto e que eles concordem que o software esteja o mais testável possível até mesmo antes de você escrever

uma única linha de código.

> **PONTO-CHAVE** A equipe de QA não é a única dona nem a guardiã da "qualidade". A responsabilidade é de todos.

Para que o nosso software tenha qualidade e para garantir que as pessoas trabalhem bem em conjunto, todos os desenvolvedores devem conhecer o processo de QA e apreciar seus detalhes intrincados.

Lembre-se de que o departamento de QA faz parte da *mesma* equipe; eles não são uma facção rival. Devemos promover uma abordagem holística e manter relacionamentos saudáveis para desenvolver um software saudável. *Tudo que precisamos é de amor.*

Perguntas

1. Qual nível de proximidade você acha que há em seus relacionamentos profissionais com os colegas de QA? Isso deveria ser melhor? Em caso afirmativo, quais passos *você* pode dar para melhorar?
2. Qual é o maior impedimento para a qualidade de software em sua estrutura de desenvolvimento? O que é necessário para corrigir isso?
3. Qual é o nível de saúde de seus procedimentos de disponibilização de versão? Como isso pode ser melhorado? Pergunte à equipe de QA o que lhe seria de mais ajuda.
4. Quem é responsável pela "qualidade" de seu software? Quem é o "culpado" quando algo dá errado? Quão saudável é isso?
5. Quão boas você acha que são suas habilidades de testes? Quão metodicamente você testa uma porção de código em que você está trabalhando antes de fazer check-in ou de passá-lo adiante?
6. Quantas falhas tolas você deixou escapar em seu código recentemente?
7. O que você pode adicionar ao seu regime de desenvolvimento *além dos testes de unidade* para garantir a qualidade do software que você entrega para a equipe de QA?

Veja também

- *É hora de testar* (Capítulo 11) – Testes de desenvolvimento – Criação de testes automatizados de unidade, de integração e de sistema.
- *O poder das pessoas* (Capítulo 34) – Trabalhar bem com uma equipe excelente de QA é um exemplo dos relacionamentos profissionais importantes que devemos promover.
- *Por favor, libere a versão* (Capítulo 23) – O processo de testes/QA e a equipe de QA são vitais para uma disponibilização eficiente de versões de software.

> **TENTE ISTO...** Comprometa-se em trabalhar mais de perto com o seu departamento de QA. Ajuste seus relacionamentos profissionais com eles para que vocês possam criar juntos um software melhor.

---

1 Esse modelo normalmente é atribuído a Winston Royce. Embora ele tenha escrito sobre ele por volta dos anos 70, o modelo era uma ilustração de um processo de desenvolvimento *problemático*, e não de um processo louvável. Veja Winston Royce, "Managing the Development of Large Software Systems" (Administrando o desenvolvimento de sistemas de software de grande porte), *Atas do* IEEE WESCON 26 (1970): 1–9.

2 Melvin E. Conway, "How Do Committees Invent?" (Como os comitês inventam?) *Datamation*

14:5 (1968): 28–31.

CAPÍTULO 22

O curioso caso do código congelado

Lá está ela esguichando! Lá está ela esguichando!
Uma corcova parecida com um monte de neve! É Moby Dick!

– Herman Melville
Moby Dick

Os gerentes falam disso em reuniões de planejamento. Os desenvolvedores o pronunciam com profunda reverência. Cerimônias de processos se desenvolvem ao seu redor. E eu tenho de me conter para não regurgitar.

É um grito que imagino vindo de um marinheiro em *Moby Dick*. Não o que diz "*lá está ela esguichando!*", mas *"congelamento do código!"*. É tão provável e igualmente tão fictício quanto o primeiro.

É nossa perseguição por outro estado de código mítico.

Atrás do congelamento de código

Congelamento de código é um termo usado por aí, presumivelmente, com boas intenções. Porém, com frequência, as pessoas não pretendem dizer o que as palavras realmente implicam.

Um congelamento de código denota o período entre algum ponto de "término" – quando nenhum trabalho adicional é esperado – e a data da disponibilização da versão.

Exatamente quando ocorrem esses pontos? E o que acontece no meio?

A *data de disponibilização de versão* é fácil de ser definida: às vezes, ela é chamada de RTM (Release to Manufacture, ou *Disponibilização para produção*). É quando o disco de instalação *Gold Master* é gravado e enviado para ser duplicado. No iluminado século XXI, nem sempre enviamos uma mídia física, porém, apesar disso, temos a tendência de

seguir as convenções mecânicas determinadas por um cronograma de disponibilização de versão desse tipo. Isso é útil e apropriado? Às vezes é, outras não. Esse processo confere uma cadência útil ao cronograma de entregas.

Entretanto qual é o "*ponto de término*" anterior que inicia o congelamento de código? Está claro que deve ser o ponto em que consideramos o código completo, com todos os recursos implementados e nenhum bug extremamente evidente. Entretanto algumas pessoas "congelam" seus códigos quando:

- Um *recurso está completo*, ou seja, quando todas as funcionalidade foram implementadas, porém não estão totalmente testadas e nenhum bug está necessariamente sendo tratado.
- A primeira *versão alfa ou beta for disponibilizada* (é claro que a definição desses estados também é maravilhosamente ambígua).
- Um build que seja *candidato a uma versão a ser disponibilizada* é gerado.

Durante esse período, "congelamos" o código para que nenhum trabalho adicional seja executado nele. No entanto essa noção é *bobagem pura*; o código jamais permanece imutável. Independentemente do que ocorrer com o código, essa é a fase em que um teste final e exaustivo de regressão é executado no software para garantir que ele está adequado para ser disponibilizado.

> **PONTO-CHAVE** "Congelamento de código" é o período que conduz a uma disponibilização de versão, quando nenhuma alteração está prevista.

No melhor caso, *congelado* é um termo figurativo. O código é considerado congelado para trabalhos de desenvolvimento, porém continua à disposição para testes finais. *Prevemos* que algumas alterações serão feitas como consequência desses testes – se não fosse possível modificar o código de modo algum, poderíamos simplesmente disponibilizá-lo *agora*, apesar de tudo.

Como estamos efetuando testes para identificar problemas, é provável que descobriremos alguns trabalhos ruins que deverão ser corrigidos. O que

acontece então? Você deve corrigir as falhas, o que significa que o código, afinal de contas, não está assim tão congelado! Não é um congelamento muito intenso.

No pior caso, a metáfora do congelamento de código não é particularmente útil. É um erro de nomenclatura. Até mesmo as geleiras se movem; apenas o fazem lentamente.

> **PONTO-CHAVE** “Congelamento de código” é um termo que pode confundir. O código jamais permanece imutável, mesmo que você queira.

Uma nova ordem mundial

Durante o congelamento do código, prevemos que alguns trabalhos finais serão necessários. Porém monitoramos cuidadosamente o desenvolvimento de software, incluindo ou excluindo alterações seletivamente no código da versão a ser disponibilizada.

Em vez de impedir totalmente as mudanças, “congelamento do código” realmente quer dizer que uma nova regra está em vigor no que diz respeito ao esforço de desenvolvimento. As alterações não podem ser aplicadas cegamente. Até mesmo as modificações que valem a pena fazer devem ser adicionadas após um consenso criterioso.

Trabalhamos arduamente para manter a integridade da versão, portanto cada mudança deverá ser cuidadosamente revisada antes de ser incluída. Incluímos somente as mudanças que sejam estritamente necessárias à versão. Nem todos os problemas ou bugs encontrados no código “congelado” serão considerados para serem corrigidos após o congelamento do código. Somente os problemas que fizerem o “show parar” – que impeçam uma versão de ser disponibilizada – serão tratados. Alguns problemas de baixa prioridade poderão ser inseridos na fila para uma próxima versão, de acordo com suas prioridades. Devemos pesar os riscos: talvez seja mais importante disponibilizar o produto em vez de investir tempo e energia para encontrar e corrigir essas falhas.

Especificamente, não haverá absolutamente nenhum trabalho adicional em novas funcionalidades. Nenhum bug será “corrigido” sem um acordo

prévio; priorizamos os problemas que devem ser tratados. É preciso ter disciplina; fazemos isso porque até mesmo a adição da funcionalidade mais simples possível ou a correção de um bug podem introduzir efeitos colaterais inesperados e indesejados.

Sendo assim, essa fase do desenvolvimento não é tanto um "congelamento" de código; é mais uma desaceleração bem intencional. É uma redução consciente da taxa de alterações das linhas de código.

> **PONTO-CHAVE** Reduzimos a velocidade do trabalho de desenvolvimento para conduzir cuidadosamente uma linha de código para a disponibilização de versão, administrando atenciosamente as últimas correções e alterações.

Uma velocidade descontrolada custa linhas (de código).

Durante o período de congelamento, algumas empresas maiores (e mais divididas em departamentos) acionam os serviços da "equipe de instalação" para criar os sistemas de instalação/distribuição ou começam a trabalhar em qualquer recurso adicional remanescente (trabalho de arte, arquivos texto etc.) para a versão final. Pessoalmente, acredito que isso esteja errado – quando entrar em "congelamento", *todo* o trabalho já deverá estar concluído e pronto para o teste final.

Formas de congelamento

Considerar as três formas diferentes de "congelamento" e sermos específicos quanto ao termos usados pode ser útil. O termo *congelamento do código* propriamente dito é um pouco confuso e enganoso:

Congelamento de funcionalidade

Um congelamento de funcionalidade declara que somente commits referentes a correções de bug poderão ser realizados – nenhuma funcionalidade nova será desenvolvida. Isso ajuda a evitar um "deslizamento de funcionalidade" – à medida que nos aproximamos do prazo para a disponibilização da versão, é sempre tentador querer incluir aquela pequena funcionalidade extra sem considerar totalmente o risco ou os bugs em potencial que a alteração poderá

introduzir.

Congelamento do código

Não trabalhamos mais em nenhuma funcionalidade nem em qualquer bug que não tenha uma prioridade bem alta. Aceitamos correções somente de problemas que realmente façam o "show parar". Precisamos urgentemente de um nome melhor e menos ambíguo para esse estado.

Congelamento de código "hard"

Nenhuma alteração é permitida. Qualquer alteração necessária após esse ponto equivale a trazer os desfibriladores e tentar ressuscitar a equipe de desenvolvimento. Jamais consideramos realmente esse estado, pois, quando chegarmos a esse ponto, o software já terá sido lançado e a equipe já terá prosseguido para outras linhas de código.

Branches são apropriados

Normalmente, quando um congelamento de código é declarado, criamos um *branch* no sistema de controle de versões. Especificamente, criamos um *branch de release* (release branch). Isso permite que a linha de desenvolvimento da versão seja congelada, sem provocar atrasos em outros trabalhos, que poderão continuar sendo feitos no branch de código principal.

Ao trabalhar com um branch de release, é uma boa prática fazer com que *absolutamente* nenhum trabalho de código seja feito no próprio branch. O branch de release sempre deve permanecer estável, sem nenhuma alteração especulativa sendo aplicada.

Em vez disso, todo o trabalho é feito em um branch mais flexível, talvez na linha principal de desenvolvimento. Cada correção é testada e conferida nesse branch, e somente quando estiver pronta ela será *mesclada* ao branch de release. Ao fazer isso, somente códigos aceitavelmente bons serão inseridos no branch de release.

O código deve sempre fluir entre os branches *em direção* a pontos de estabilidade. Nós "promovemos" os conjuntos de alterações de acordo

com a sua qualidade comprovada.

Todas as alterações incorporadas a um branch congelado são submetidas a um tratamento mais rigoroso que as alterações anteriores de desenvolvimento:

- são cuidadosamente revisadas;
- recebem esforços de teste focados;
- são analisadas quanto ao risco, de modo que qualquer diferença em potencial que introduzirem sejam bem compreendidas e, se for necessário, os riscos sejam atenuados;
- são priorizadas – as alterações serão cuidadosamente revistas para verificar se são apropriadas para a versão.

Os branches são cruciais para uma equipe ser capaz de administrar congelamentos de código. Sem um branch de release, todos os desenvolvedores teriam de parar de usar fisicamente as suas ferramentas e interromper seus trabalhos durante o congelamento. Não seria um bom uso de tempo e de recursos caros. Os desenvolvedores gostam de desenvolver; logo eles ficariam irrequietos e começariam a escrever códigos, de qualquer maneira.

PONTO-CHAVE Crie um branch ou você terá problemas.

Apesar do que foi dito, é uma boa ideia evitar trabalhos concorrentes o máximo possível – pode ser confuso e resultar em metas conflitantes para a equipe.

Mas o código não está realmente congelado!

Tome cuidado para que o termo "congelamento de código", que é inadequado, não conduza você a um falso sentimento de autoconfiança. Com frequência, o termo congelamento de código é proferido aos gerentes para indicar um estado mais estável do projeto e conquistar-lhes a confiança. *Soa* muito bem, não é mesmo?

Porém não acredite que o seu código estará em um estado melhor do que está. A todo momento, é importante ter uma avaliação realista do estado

de seu projeto.

Tome cuidado para que a palavra “congelamento” não faça você cair na tentação de manter a situação rígida quando isso não deveria ocorrer. Quando forem necessárias, as alterações *deverão* ser feitas.

Duração do congelamento

Você deve declarar um inverno digital pelo período de tempo correto. Como no inverno de Nárnia, você não vai querer um congelamento desnecessariamente longo, em que o Natal jamais chegue! Mas, se esse período for curto demais, o congelamento será um exercício sem sentido.

O período correto depende da complexidade de seu projeto, das exigências de teste que ele impõe (tanto em relação às pessoas quanto aos recursos: devemos instalar ou configurar uma plataforma de teste totalmente separada, com administradores e especialistas para mantê-la em execução?), do escopo das alterações incluídas nessa versão (quais podem influenciar o nível do teste de regressão executado) e dos recursos disponíveis a serem dedicados aos testes e à verificação.

Um período típico de congelamento dura duas semanas.

Tome cuidado com o princípio de Pareto: com frequência, nós o vemos em projetos de TI em que os “últimos” 20% de esforço se expandem até atingir 80% do tempo total (ou algo por aí). Para evitar isso, certifique-se de entrar no período de congelamento no momento certo. Não declare um congelamento quando achar que você só precisar “terminar” algumas coisinhas. Faça o congelamento quando tudo *estiver* concluído.

Sinta o congelamento

Um congelamento do código é o caminho difícil para disponibilizar uma versão – e não um piquenique no parque. Defina suas expectativas de acordo com isso.

Durante um período de congelamento de código, espere encontrar bugs que você *não poderá corrigir* porque eles não serão importantes o suficiente para assumirmos o risco de serem incluídos. Esse não é mais

um código livre para todos, em que qualquer alteração seja permitida; se não fosse assim, você não teria declarado o seu congelamento. Desse modo, espere ficar desapontado e lançar um produto que você esperaria que fosse melhor!

> **PONTO-CHAVE** Não é incomum (nem errado) lançar um software que você sabe que poderia ser melhor.

Veja pelo lado bom: como esses bugs foram encontrados, eles poderão ser corrigidos na próxima versão.

Espere também acumular *débitos técnicos* durante um congelamento.[1] Esse é um dos poucos momentos válidos para fazer isso: quando não houver escopo para fazer correções amplas, os problemas deverão ser corrigidos com técnicas de remendos à base de "papel sobre fissuras" para ter um produto "bom o suficiente" para ser lançado. Contudo lembre-se de considerar esse tipo de trabalho como *débito*, e não como uma prática normal, e planeje pagá-lo nos ciclos de desenvolvimento após a disponibilização da versão.

> **PONTO-CHAVE** Durante o congelamento de código, você contrairá débitos técnicos. Monitore isso e esteja preparado para pagar a dívida assim que sua versão for lançada.

Se você fizer uma alteração que tenha sérias implicações durante o congelamento, considere se o código não deverá ser descongelado e congelado novamente, com um ciclo completo de testes iniciado desde o início. Adie a disponibilização da versão e reinicie o seu período de congelamento de código se for necessário.

Os cientistas dizem que congelar, descongelar e congelar novamente não é bom para a sua saúde. Portanto tome cuidado para não fazer isso muitas vezes, ou você acabará tendo uma intoxicação alimentar!

Um período longo de congelamento é um indicador de que você não tem uma base de código estável o suficiente.

O fim se aproxima

No final de um período de congelamento de código, quando atingimos o

ponto de RTM, a linha de código estará *realmente* congelada. Nenhuma alteração deve ocorrer agora, pois a versão (finalmente) foi disponibilizada. Feche o branch de release. Arquive a linha de código. Vá em frente e comemore.

Qualquer alteração adicional será feita em uma linha de código diferente.

> **PONTO-CHAVE** O único "congelamento de código" verdadeiro ocorre quando uma versão aceita é disponibilizada. Esse é o ponto em que o código é finalmente *gravado a ferro e fogo*.

Esse é o ponto em que temos um *verdadeiro* congelamento de código. Porém ninguém jamais fala sobre isso!

Anticongelamento

Se você trabalhar bem, será possível evitar totalmente os períodos de congelamento de código. Essa dança sórdida *pode* ser ignorada.

Muitas equipes de desenvolvimento não estão mais limitadas por um processo físico de produção – elas lançam o software pela Internet ou criam web services que podem ser implantados em servidores de produção em um piscar de olhos. O "desastre" de um bug passar e chegar até uma versão externa é minimizado nesse caso – uma atualização de software online pode ser feita para remediar o problema em campo antes que muitos usuários sequer identifiquem o problema.

Entretanto essa não é nenhuma desculpa para sair jogando um código por aí sem efetuar testes. Versões rápidas sem períodos de congelamento de código exigem uma nova mentalidade, além de disciplina. Nós nos esforçamos para disponibilizar uma versão rapidamente ao escrever um código confiável, comprovadamente livre de bugs, desde o início.

Podemos minimizar e até mesmo eliminar os períodos de congelamento de código ao:

- Realizar entregas contínuas; devemos definir um fluxo que leve cada build a um estado totalmente passível de implantação. Isso garante que você estará sempre pronto para uma implantação.

- Estabelecer uma boa estrutura de testes *automatizada*, com uma boa cobertura. Esses testes devem abranger o código, a integração e os aspectos finais voltados aos usuários do sistema, de modo a obter um feedback confiável sobre o estado do produto.
- Bons critérios de aceitação de testes – ferramentas como o Cucumber (*http://cukes.info/*) podem ser usadas para garantir que o conjunto completo de requisitos de alto nível de usuários tenha sido atendido pelo software.
- Reduzir o período de testes – reduzir o escopo ou o tamanho do projeto para que você não precise de um período longo de travamento para cada versão.
- Desenvolver um "fluxo de disponibilização de versão" simples e confiável que conduzirá o seu código para a produção e o fará ser implantado com poucos esforços e sem nenhuma intervenção humana.

Com esse tipo de disciplina, é perfeitamente possível "disponibilizar" um código para a produção regularmente, com muito menos formalidades do que no processo de engenharia tradicional para disponibilização de versões. Muitas equipes podem disponibilizar uma versão de software a cada semana. Algumas são capazes de implantar o código em seus sistemas de produção diariamente.

Esses ciclos extremamente curtos de desenvolvimento exigem uma mentalidade de codificação muito mais disciplinada ao longo do processo, portanto não será preciso mudar de marcha e ter mais cuidado no período de congelamento da fase de desenvolvimento.

> **PONTO-CHAVE** Tenha como meta ter um código que jamais seja "congelado" e que esteja permanentemente pronto para ser disponibilizado para produção.

Conclusão

Congelamento de código é um termo problemático; é uma metáfora que confunde. O código não é realmente congelado nem descongelado. É uma substância maleável, que muda e se adapta constantemente ao mundo ao

seu redor. O que realmente acontece é que a taxa de mudanças no desenvolvimento se torna mais lenta e mudamos o foco de nosso trabalho.

Entretanto é verdade que, à medida que nos aproximamos de uma disponibilização de versão, devemos ter mais disciplina no regime de desenvolvimento para garantir que o software tenha qualidade suficiente para ser disponibilizado.

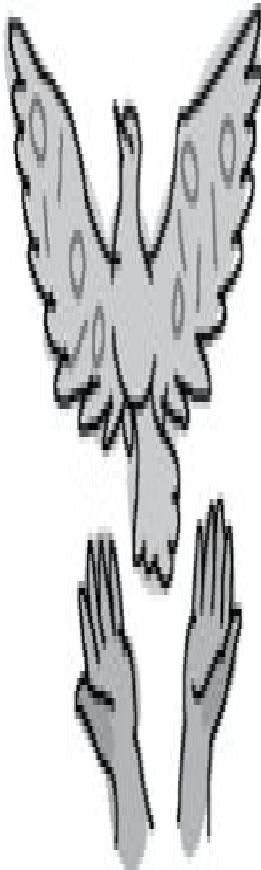
Perguntas

1. Você tem um período formal de congelamento de código em sua prática de desenvolvimento? Com que zelo ele é observado?
2. Como você garante que as alterações aplicadas no período de congelamento sejam seguras e apropriadas?
3. A qualidade do build é responsabilidade de uma só pessoa ou de toda a equipe? Qual é a abordagem correta e por quê?
4. Seu projeto precisa de um período de tempo longo para atingir o ponto de congelamento de código? Por quê? Como esse período pode ser reduzido?

Veja também

- *Por favor, libere a versão* (Capítulo 23) – Um “congelamento de código” existe para ajudar você a estabilizar o seu software na corrida para uma disponibilização de versão.
- *Controle eficiente de versões* (Capítulo 20) – Um *branch de release* é usado para encapsular um código congelado.

TENTE ISTO... Considere o modo de melhorar o seu processo de disponibilização de versão. Como você pode minimizar (ou eliminar) o período de congelamento de código?

1 Para mais informações sobre a metáfora do *débito técnico*, acesse *http://martinfowler.com/bliki/TechnicalDebt.html*.

CAPÍTULO 23

Por favor, libere a versão

Ouvi um anjo cantando
quando o dia estava raiando,
"Perdão, Compaixão, Paz
Liberdade ao mundo traz."

– William Blake
Ouvi um anjo

Criar uma *versão de software* é um passo extremamente importante em seu processo de desenvolvimento de software e é uma tarefa que não deve ser deixada para a última hora. Ela exige disciplina e planejamento.

Muitas vezes, já me deparei com problemas tolos, causados por uma abordagem precária na geração de versões de software, que poderiam perfeitamente ter sido evitados.

A maior parte desses problemas era causada pelo péssimo hábito de criar uma “versão” a partir de um diretório de trabalho local em vez de fazê-lo a partir de um checkout limpo. (Dica: essa não é uma verdadeira disponibilização de versão de software, mas um “build” de seu código; é preciso *muito* mais processo e diligência para criar uma versão adequada.)

Alguns exemplos:

- Uma versão de software foi criada a partir do diretório de trabalho local de um desenvolvedor. O desenvolvedor não havia feito uma limpeza no código antes e o diretório continha alterações em códigos-fonte para as quais não havia sido feito um commit. Ele percebeu isso, porém gerou a “versão” de qualquer modo. Quando os problemas

foram relatados, não tínhamos nenhum registro exatamente *do que* havia sido incluído nesse build. Resultado: depurar o software foi um pesadelo – um trabalho de adivinhação, em sua maior parte.

- Uma versão de software havia sido criada a partir de um diretório local que não estava atualizado; o desenvolvedor não o havia atualizado de acordo com o HEAD (versão mais recente) do repositório de código do Subversion. Desse modo, havia funcionalidades e correções de bug faltando na versão. Para ajudar, o desenvolvedor havia atribuído a tag "release point" (ponto de disponibilização de versão) ao HEAD do repositório, argumentando que havia gerado aquela versão. Resultado: muita confusão, constrangimento e uma mancha na reputação do projeto.
- Uma versão do projeto havia sido disponibilizada, cujo código não estava em um sistema de controle de versões; o projeto estava no disco rígido local de um computador e – tenha certeza de que você irá adivinhar – seu backup não estava sendo feito. O código era baseado em uma versão de software original criada por outra empresa. Não tínhamos nenhum registro da proveniência do código original, nem sabíamos em que versão ele estava baseado ou como ele havia sido subsequentemente modificado. Para piorar, o ambiente exato de build do computador era desconhecido – resultado de anos de acertos e de ajustes. A lei de Murphy entrou em ação. O computador parou de funcionar. Resultado: todas as cópias do código-fonte foram perdidas, juntamente com o conhecimento de como gerar um build para a plataforma-alvo. Fim de jogo.

Cada uma dessas experiências foi incrivelmente frustrante.

Criar uma versão de software séria e de alta qualidade representa muito mais trabalho do que simplesmente selecionar a opção "build" em seu IDE e enviar ao cliente o que quer que tenha sido gerado. Se você não estiver preparado para investir nesse trabalho extra, *não* gere versões.

> **PONTO-CHAVE** Gerar versões de software exige disciplina e planejamento. É mais do que simplesmente selecionar a opção "build" no IDE de um desenvolvedor.

No capítulo anterior, demos uma olhada no "congelamento de código", em que tentamos estabilizar o nosso código na corrida para a disponibilização de uma versão. Também questionamos se um período de congelamento de código deveria existir, antes de tudo. Depois que o seu código estiver pronto para a disponibilização de uma versão, seja na forma de uma imagem de sua linha principal de desenvolvimento ou de um branch congelado, você deverá ter rigor e disciplina para gerar uma versão sólida a partir daí.

Parte do processo

A maioria das pessoas escreve software tanto para os outros quanto para si mesma. Sendo assim, ele deverá chegar às mãos de seus "usuários" de alguma maneira. Independentemente de você acabar gerando um instalador de software em um CD, um instalador que possa ser baixado, um arquivo zip com os códigos-fontes ou de implantar o software em um servidor web ativo, essa tarefa corresponde ao importante processo de *disponibilização de uma versão de software*.

O processo de disponibilização de versão é uma parte crucial de seu regime de desenvolvimento de software, tão importante quanto o design, a codificação, o debugging e os testes. Para ser eficiente, seu processo de disponibilização de software deve ser:

- simples
- repetível
- confiável

Faça isso da forma errada e você estará acumulando problemas desagradáveis em potencial para o seu próprio futuro. Ao gerar uma versão, você deve:

- Garantir que poderá obter exatamente o mesmo código que a gerou a partir de seu sistema de controle de versões. (Você utiliza um sistema de controle de versões, certo?) Essa é a única maneira concreta de provar quais bugs foram e quais não foram corrigidos nessa versão. Dessa maneira, quando for necessário corrigir um bug crítico na

versão 1.02 de um produto que tenha cinco anos de idade, você poderá fazê-lo.

- Registre exatamente como a versão foi gerada (incluindo as configuração de otimização do compilador, a configuração da CPU-alvo etc.). Esses recursos poderão afetar sutilmente a maneira como o seu código irá executar e ajudar a verificar se determinados bugs se manifestam.
- Armazene o log de build para futuras referências.

Uma engrenagem na máquina

Um esquema básico de um bom processo de disponibilização de versão, especificamente para uma aplicação "shrink-wrap" (embalada) que possa ser distribuída, está sendo mostrado a seguir.[1]

Passo 1: inicie a disponibilização de versão

Concorde que é hora de iniciar a disponibilização de uma nova versão. Uma disponibilização formal de versão é tratada de modo diferente de um build de teste de um desenvolvedor e *jamais* deverá ser feita a partir de um diretório de trabalho existente.

Chegue a um consenso em relação ao nome e ao tipo da versão (por exemplo, "5.06 Beta1" ou "1.2 Candidato à versão").

Passo 2: prepare a versão

Determine exatamente qual código comporá essa versão. Na maioria dos casos, você já estará trabalhando em um *branch de release* em seu sistema de controle de versões, portanto é o estado desse branch nesse momento. Raramente, você deverá gerar uma versão diretamente do código da linha principal de desenvolvimento de seu sistema de controle de versões (por exemplo, *tronco* ou *master*).

Um branch de release corresponde a uma imagem estável do código que permite que você continue o desenvolvimento de funcionalidades "instáveis" no tronco. Um trabalho que seja bom, estável e que funcione

comprovadamente pode ser mesclado da linha principal para o branch de release depois que estiver testado. Isso mantém a integridade da base de código da versão ao mesmo tempo que permite que novos trabalhos continuem sendo feitos na linha principal de desenvolvimento.

Se você executar testes de unidade e um sistema de integração contínua na linha principal de seu código, então programe-se para executá-los também no branch de release enquanto ele estiver ativo. O processo de disponibilização de versão deve ser breve e, desse modo, os branches de release devem ter vida curta.

Aplique uma *tag* no sistema de controle de versões para registrar o que está sendo incluído na versão. O nome da tag deve refletir o nome da versão.

Branches de release

Você pode criar um branch em seu sistema de controle de versões praticamente por qualquer motivo: para trabalhar em uma nova funcionalidade de modo independente do restante da base de código, para trabalhar em uma correção de bug ou para efetuar uma refatoração exploratória, sem desestruturar outros códigos. O branch engloba um conjunto de alterações. Ao concluir o trabalho, normalmente você irá mesclar o branch de volta à linha principal de desenvolvimento. Ou irá jogá-lo fora. É padrão.

"Branches de release" são criados exatamente pelo motivo oposto – para assinalar um ponto na base de código em que você deseja garantir *estabilidade*. Depois de ter criado um branch para a sua versão de software, outras funcionalidades novas poderão ser desenvolvidas na linha principal de código sem a preocupação de comprometer a disponibilização de versão pendente. Isso provê um mecanismo para efetuar o seu "congelamento de código".

O departamento de QA pode efetuar testes de regressão do código no branch sem se preocupar com o fato de novos trabalhos alterarem a posição do gol bem debaixo de seus pés. Ocasionalmente, uma correção de bug importante na linha principal de desenvolvimento poderá ser mesclada ao branch de release, porém cada mescla será feita com muito cuidado, com o consentimento de todas as pessoas que tiverem um interesse na qualidade do software que está nesse branch.

Um branch de release não é mesclado de volta ao tronco, pois nenhum desenvolvimento novo é feito aí. Todo o desenvolvimento ocorre no tronco, é testado e, em seguida, mesclado no branch de release.

Um bom processo de disponibilização de versão não exige necessariamente um branch de release. Se você mantiver a linha principal de seu código em um estado permanentemente passível de ser disponibilizado em uma versão (ao desenvolver com mais rigor e empregando testes automatizados sólidos), será possível ignorar totalmente o ritual de criação de um branch de release.

Passo 3: gere a versão

Faça checkout de uma cópia virgem de toda a base de código que tenha a tag correspondente. *Jamais* use um checkout já existente. Você poderá ter alterações locais para as quais um commit não tenha sido efetuado e que afetarão o build, ou outros arquivos não controlados que poderão fazer diferença. Sempre aplique uma tag e, *em seguida*, faça o checkout usando essa tag. Isso evitará muitos problemas em potencial.

PONTO-CHAVE Sempre gere o build de seu software a partir de um checkout recente, feito do zero. Jamais reutilize partes antigas de um build de software.

Agora faça o build do software. Esse passo *não deve* envolver a edição manual de nenhum arquivo, pois, do contrário, você não terá um registro do código exato a partir do qual o build foi feito.

O ideal é que o build seja *automatizado*: deve ser feito por meio de um único botão pressionado ou pela chamada de um único script. Efetuar check-in da mecânica do build no sistema de controle de versões, juntamente com o código, registra o modo como o código foi construído de forma não ambígua. A automação reduz o potencial de erros humanos no processo de disponibilização de versão.

PONTO-CHAVE Deixe o seu build simples, com um passo único que automatize todas as partes do processo. Utilize uma linguagem de scripting para isso.

O script de build atua como uma documentação não ambígua de como é feito o build do projeto. Ele garante também que seja simples implantar o build em um servidor de CI (*integração contínua*), em que ele poderá ser

executado automaticamente para garantir a sua validade. Com efeito, os melhores builds de versão são acionados diretamente de um servidor de CI e jamais entram em contato com mãos humanas.

> **PONTO-CHAVE** Implante seus builds em um servidor de CI para garantir a sua saúde. Gere versões formais a partir do mesmo sistema.

Passo 4: empacote a versão

O ideal é que essa seja uma parte integrante do passo anterior.

Componha um conjunto de "informações da versão" (release notes) que descrevam como a versão difere da versão anterior: as novas funcionalidades e os bugs que foram corrigidos. (Isso pode ser automatizado por meio da extração de informações dos logs do sistema de controle de versões. Entretanto as mensagens de check-in podem não ser as melhores informações a serem apresentadas aos clientes.)

Empacote o código (crie uma imagem para a instalação, imagens de CD ISO etc.). Esse passo também deve fazer parte do script automatizado de build pelo mesmo motivo.

Passo 5: implante a versão

Armazene os artefatos gerados e o log de build para referências futuras. Coloque-os em um servidor de arquivos compartilhado.

Teste a versão! Sim, você já testou o código para provar que era o momento de disponibilizar uma versão, porém, agora, você deve testar essa versão e garantir que tudo esteja correto e que os artefatos tenham uma qualidade adequada para serem disponibilizados.

> **PONTO-CHAVE** Jamais disponibilize um software sem testar os artefatos finais.

Deve haver um teste de fumaça (smoke test) inicial para garantir que os instaladores funcionem bem (em todas as plataformas de implantação suportadas) e que o software pareça executar e funcionar corretamente.

Em seguida, execute qualquer teste que seja apropriado para esse tipo de disponibilização de versão. Versões para testes internos podem passar por scripts de teste executados pelo pessoal interno de testes. Versões beta

podem ser disponibilizadas para pessoas externas selecionadas que irão testá-las. Candidatos a versões oficiais devem ser submetidos a verificações finais de regressão adequadas. Uma versão de software não deverá ser distribuída até que tenha sido totalmente verificada.

Por fim, distribua a versão. Se você enviá-la diretamente a um usuário final, isso talvez envolva colocar um instalador em seu site e enviar emails ou notas para a imprensa. Pode ser que signifique enviar o software para uma empresa para que sejam gravadas mídias físicas.

Se o seu código for implantado em um servidor de produção ativo em execução, estaremos entrando no mundo do *devops* (*desenvolvimento/operações*, em que o desenvolvimento de software se entrelaça com operações de TI). Essa é a arte de implantar um código novo em um servidor remoto, atualizando qualquer componente de software necessário, atualizando sistemas de armazenamento de dados (por exemplo, executando migrações de esquemas de banco de dados) e reiniciando o servidor com o código novo; tudo com o menor período de inatividade (downtime) possível. Esse processo deve contar com um fallback sensato caso haja algum problema de instalação. Como ocorre com vários outros processos de software, a automação é o principal segredo para uma implantação eficiente e bem-sucedida em um servidor.

Disponibilize versões cedo e com frequência

Um dos piores pecados do processo de disponibilização de versão é pensar nesse assunto somente quando o final do projeto for atingido, quando você finalmente tiver de efetuar uma disponibilização pública de software.

No mundo do software, adiar qualquer tarefa ou decisão até o *último momento responsável* é uma crença cada vez mais popular[2]; isso que dizer adiá-la até o ponto em que você souber o máximo sobre os requisitos exatos e tenha minimizado o custo de oportunidade em relação ao que você poderia estar fazendo no lugar disso. É um ponto de vista sólido; contudo o último minuto responsável para criar um processo de disponibilização de versão é muito mais cedo do que a maioria dos

desenvolvedores espera.

> **PONTO-CHAVE** Não adie o planejamento e a criação de um processo de disponibilização de versão de software deixando-o para o último instante. Crie-o cedo, faça iterações pelos builds de forma rápida e frequente. Depure o build como você depuraria qualquer outro software.

Já vimos que o processo ideal de disponibilização de versão é totalmente automatizado. O sistema de build e disponibilização de versão automatizado deve ser estabelecido bem *cedo* no processo de desenvolvimento. Ele deve então ser usado com frequência (diariamente, se não com mais frequência) para provar que funciona e que é robusto. Isso ajudará também a enfatizar e a eliminar qualquer pressuposição nefasta no código sobre o ambiente de instalação (paths fixos no código, suposições sobre as bibliotecas instaladas no computador, recursos etc.).

Iniciar o processo de disponibilização de versão de software cedo no trabalho de desenvolvimento lhe dará bastante tempo para aparar todas as arestas e falhas de modo que, quando você estiver na corrida para uma verdadeira disponibilização de versão para o público, seja possível focar na importante tarefa de fazer o seu software funcionar, em vez de se concentrar no mecanismo tedioso de descobrir como efetuar o seu lançamento. Com efeito, algumas pessoas definem seus processos de disponibilização de versão como a primeira tarefa em um novo projeto de software.

E tem mais...

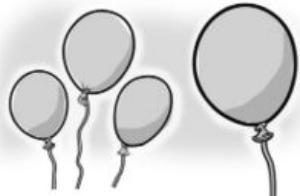
Esse é um assunto amplo, intimamente relacionado ao gerenciamento de configuração, ao controle de códigos-fontes, aos procedimentos de testes, ao gerenciamento de produtos de software e a outras atividades desse tipo. Se fizer parte do processo de disponibilização de um produto de software, você realmente deverá entender e respeitar a santidade do processo de disponibilização de versões de software.

Perguntas

1. Em que momento você decide que deve iniciar uma nova

disponibilização de versão de software?

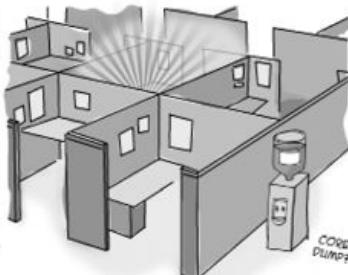
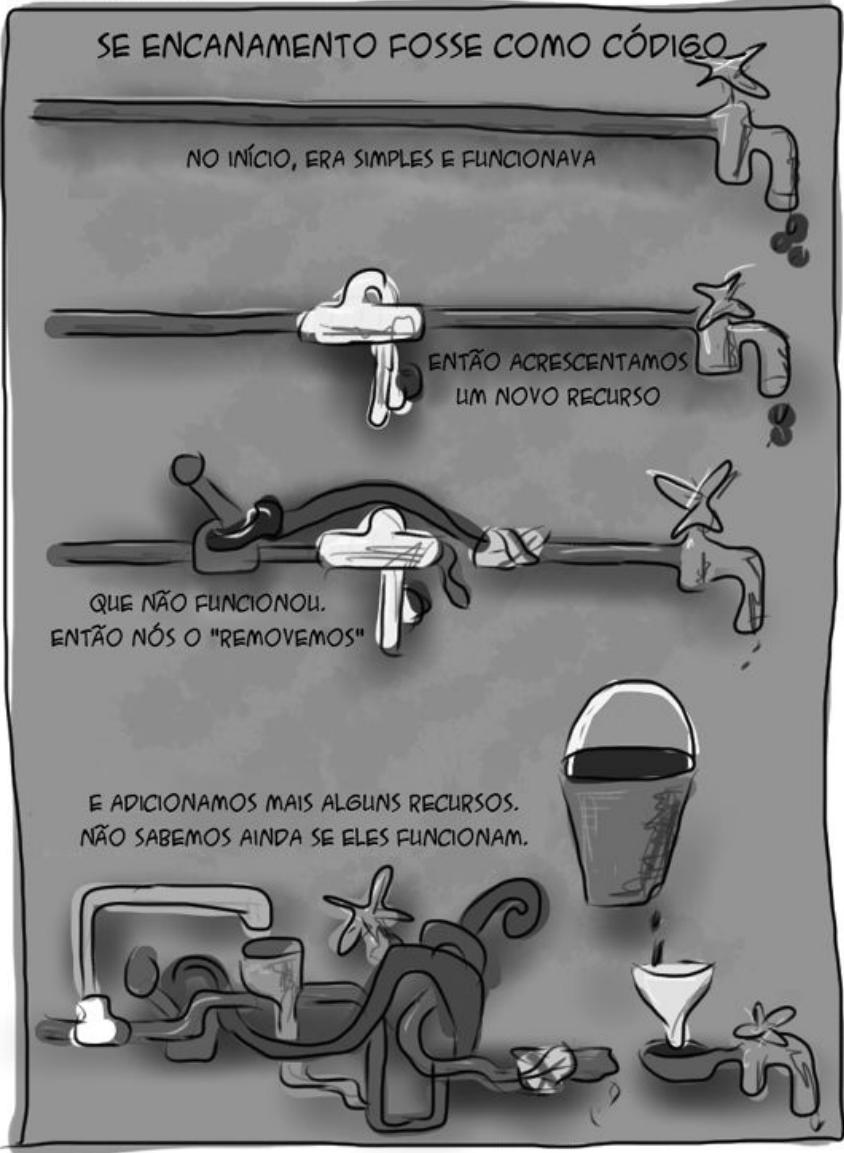
2. Quão repetível e confiável é o seu processo de build e de disponibilização de versão? Quão simples ele é? Com que frequência os builds falham?
3. Qual foi a pior falha de geração de versão que você já viu? Como isso poderia ter sido evitado?
4. O seu build falha de modo intermitente? Isso ocorre nos computadores dos desenvolvedores ou no servidor de CI? Qual é o pior lugar para isso ocorrer?
5. A criação e o amadurecimento do processo de build e de disponibilização de versão devem ser tarefas específicas de uma pessoa específica ou todos da equipe devem ser responsáveis por isso? Por quê?

Veja também

- *O curioso caso do código congelado* (Capítulo 22) – Como tentamos conduzir o trabalho de desenvolvimento de código em direção a uma disponibilização de versão.
- *Controle eficiente de versões* (Capítulo 20) – Uma versão de software é gerada a partir de código virgem obtido diretamente do repositório de códigos-fontes. O SCV armazena nosso *branch de release* do código.

TENTE ISTO... Revise o processo de build e de disponibilização de versão de seu projeto. Descubra como melhorá-lo. Se ele não estiver automatizado por meio da chamada a um único script, implemente esse script agora.

10.000 MACACOS
(OU ALGO POR AÍ)

POR FAVOR, LIBERE A VERSÃO

ESQUEÇA A LIBERAÇÃO DE VERSÃO. TENTE...

PARA GANHAR PONTOS ADICIONAIS, RABISQUE INSULTOS PARA OS EXPERTS EM ENCONTRAR FALHAS NO CÓDIGO

1 Outros tipos de disponibilização de versão – por exemplo, em um servidor web ativo – em sua maior parte, seguem esse padrão, com alguns ajustes.

2 Isso é originalmente do livro *Lean Software Development: An Agile Toolkit* (Boston: Addison-Wesley, 2003) de Mary e Tom Poppendieck, em que o termo é descrito como *o momento em que deixar de tomar uma decisão elimina uma alternativa importante*.

PARTE III

Envolvendo-se pessoalmente

Tornar-se um programador melhor exige mais que mero domínio de uma boa codificação e de um bom design, apesar de esses assuntos já serem profundos o suficiente. Há todo um conjunto de outras habilidades que você deve adquirir, além de atitudes e de abordagens que transformarão você em um mestre.

Os capítulos a seguir o ajudarão a crescer nessa área, apresentando uma seleção de assuntos sobre desenvolvimento pessoal. Aprenderemos a aprender, a considerar um comportamento ético, a ser desafiado a fim de evitar a estagnação, além de cuidar de nosso bem-estar físico. Essas partes são todas importantes na vida de um programador.

CAPÍTULO 24

Viva para amar o aprendizado

Aprender é como remar rio acima: deixar de avançar é recuar.

– Provérbio chinês

A programação é um campo empolgante e dinâmico para trabalhar; *sempre* há algo novo para aprender. Os programadores raramente são forçados a repetir a mesma tarefa durante anos e sempre descobrem novas maneiras de desenvolver LER e perder a visão. Estamos continuamente encarando o desconhecido: novos problemas, novas situações, novas equipes, novas tecnologias ou uma nova combinação de tudo isso.

Somos continuamente desafiados a aprender, a melhorar nossas habilidades e nossas capacidades. Se você achar que sua carreira está ficando estagnada, um dos passos mais práticos que pode ser dado para sair desse caminho é fazer um esforço consciente para *aprender algo novo*.

PONTO-CHAVE Mantenha-se em um estado de aprendizado contínuo. Sempre procure aprender algo novo.

Algumas pessoas são naturalmente melhores para aprender; elas se superam ao absorver novas informações e podem "ganhar velocidade" rapidamente. É natural. Porém é algo em que todos podemos nos aperfeiçoar se tentarmos. Você deve assumir a responsabilidade pelo seu aprendizado.

Se quiser melhorar como programador, você *deverá* ser um aluno habilidoso e experiente. E deve aprender a apreciar essa tarefa.

PONTO-CHAVE Aprenda a *gostar* de aprender.

O que você deve aprender?

Há um mundo de assuntos que você pode tentar escolher. Então em que você deve dar uma olhada? O poeta político norte-americano Donald Rumsfeld sintetizou esse problema de maneira particularmente adequada quando participou de uma infame coletiva de imprensa na Casa Branca:

> *Como todos sabemos, há conhecimentos conhecidos; há assuntos que sabemos que sabemos. Também sabemos que há desconhecidos conhecidos; isso equivale a dizer que sabemos que há assuntos sobre os quais não sabemos. Mas há também os desconhecidos desconhecidos – aquilo que não sabemos que não sabemos.*

Isso é admiravelmente profundo.

Você deve ter como meta o *desconhecido conhecido* – algo que você gostaria de aprender. Ou vá atrás dos *desconhecidos desconhecidos*: invista tempo inicialmente investigando assuntos interessantes para aprender.

Em última instância, escolha algo que lhe interesse. Algo que irá beneficiá-lo (o ato de aprender por si só *é* um benefício, porém escolher um assunto porque ele lhe dará novas habilidades úteis, ampliará sua visão ou lhe dará prazer é bom). Você irá investir uma quantidade de tempo significativa, portanto invista com sabedoria!

Conheça uma nova tecnologia

Para os programadores, essa é uma opção óbvia. Somos fascinados pelas diferentes maneiras pelas quais podemos fazer os elétrons dançarem, e há muitas maneiras de fazer isso.

Não faltam linguagens de programação novas e interessantes; você não precisa se tornar um especialista, mas vá além do "*Hello, World!*". Conheça uma nova biblioteca, um framework de aplicação ou uma ferramenta de software. Aprenda a usar um novo editor de texto ou um IDE. Conheça uma nova ferramenta de documentação ou um framework de teste. Conheça um novo sistema de build, um sistema de monitoração de problemas, um sistema de controle de versões (seja indulgente consigo mesmo e conheça a nova mania do *sistema distribuído de controle de versões* de que todos os garotos bacanas estão falando), um novo sistema operacional e assim por diante.

Adquira novas habilidades técnicas

Aprenda a ler um código desconhecido de modo eficiente ou a escrever uma documentação técnica. Aprenda a desenvolver a arquitetura de um software.

Aprenda a trabalhar com pessoas

Sim, isso é tediosamente "sentimental demais" para a maioria dos programadores nerds. Mas é uma área extremamente interessante e útil. Estude sociologia ou leia textos sobre gerenciamento. Leia sobre como se tornar líder de uma equipe de software. Isso ajudará você a se tornar um funcionário mais capacitado a trabalhar em equipe. Ajudará você a aprender a se comunicar bem com a sua equipe e a entender melhor o seu cliente.

Conheça um novo domínio de problema

Talvez você sempre tenha tido vontade de escrever software para modelagem matemática ou trabalhar com DSP (Digital Signal Processor, ou Processador digital de sinais) de áudio. Sem ter nenhuma experiência ou conhecimento, é improvável que você vá encontrar um emprego em uma nova área, portanto saia na frente e comece a conhecer o assunto. Em seguida, descubra um modo de adquirir experiência prática e demonstrável.

Aprenda a aprender

É sério! Descubra novas técnicas que o ajudarão a absorver conhecimentos de maneira mais eficiente. Você tem a impressão de que há um constante bombardeio de informações em que você deve dar uma olhada, mas elas parecem simplesmente fluir, passando por você? Investigue maneiras de procurar, consumir e absorver conhecimentos. Exercite novas habilidades como criar mapas mentais e fazer leitura dinâmica.

Aprenda algo totalmente diferente

Ou, mais interessante ainda, foque em uma área totalmente diferente, sem relevância para o seu trabalho cotidiano e sem nenhuma aplicação óbvia em software. Aprenda uma nova língua estrangeira, a

tocar um instrumento musical, conheça um novo ramo da ciência, da arte ou da filosofia ou até mesmo da espiritualidade. Isso ampliará a sua visão de mundo e, inevitavelmente, dará mais base ao modo como você programa.

Aprendendo a aprender

Aprender é uma habilidade humana básica. Todos nós fazemos isso o tempo todo. Depois de ser formatado, o cérebro humano absorve informações rapidamente e desenvolve habilidades por meio de uma ampla variedade de experiências.

Em seguida, passamos para o treinamento acadêmico, em que nosso aprendizado se afunila progressivamente por um sistema cada vez mais restrito. Passamos de uma educação geral para uma educação secundária mais especializada, até que focamos em um único tópico principal na universidade. Os estudos de pós-graduação focam em uma área mais específica ainda desse tópico específico. Esse foco cada vez mais restrito possibilita que nos tornemos altamente proficientes em uma área, porém, nesse processo, somos treinados para termos uma mentalidade extremamente restrita.

PONTO-CHAVE Nosso aprendizado, com frequência, é focado de maneira muito restrita. Considere uma esfera mais ampla de referência. Procure inspiração em diversos campos.

Há algumas técnicas que podem ser empregadas para ajudar você a aprender de modo a ser mais bem-sucedido.

Saiba como aprender melhor e explorar isso em benefício próprio; o seu tipo de personalidade afeta o seu estilo de aprendizado. As pessoas que são tradicionalmente classificadas como tendo o "lado direito" do cérebro mais desenvolvido aprendem melhor quando são apresentadas a padrões e a uma visão holística de um assunto; elas têm mais dificuldade quando são bombardeadas com um fluxo serial de informações. Aquelas que têm o "lado esquerdo" do cérebro mais desenvolvido, por outro lado, são mais bem-sucedidas com uma apresentação linear e racional de um assunto.

Elas preferem assimilar fatos em vez de ouvir uma ótima história.[1] Pessoas introvertidas preferem aprender por conta própria, enquanto os extrovertidos se dão muito bem em workshops em que há cooperação. Conhecer o seu tipo particular de personalidade revelará maneiras específicas de tornar a sua rotina de aprendizado o mais eficiente possível.

Você pode achar útil tentar absorver informações da várias fontes diferentes. No mundo moderno e conectado, temos várias formas de mídia à disposição:

- escrita (por exemplo, livros, revistas, blogs);
- falada (por exemplo, audiolivros, palestras, grupos de usuários, podcasts, cursos);
- visual (por exemplo, podcasts de vídeos, programas de TV, apresentações).

Algumas pessoas respondem melhor a determinadas mídias. O que funciona melhor para você? Para ter os melhores resultados, combine várias dessas fontes. Use podcasts sobre um assunto para reforçar o que estiver lendo em um livro. Participe de um curso de treinamento e leia um livro sobre o assunto também.

> **PONTO-CHAVE** Utilize o máximo possível de fontes para melhorar a qualidade de seu aprendizado.

O *feedback entre sentidos* tenta estimular partes do cérebro que normalmente não são exercitadas durante o aprendizado para melhorar a eficiência do cérebro. Considere experimentar algumas dessas ações quando estiver aprendendo – algumas poderão funcionar bem para você:

- ouça música enquanto trabalha;
- rabisque enquanto pensa (sim, estou prestando atenção em sua reunião, veja quantos rabiscos eu fiz...);
- fique mexendo em um objeto (uma caneta ou um clipe de papel, talvez);
- fale enquanto trabalha (vocalizar o que você está fazendo ou aprendendo realmente ajuda a reter mais conhecimentos);

- torne os processos de raciocínio concretos em vez de serem puramente intelectuais – como modelar usando blocos de construção ou usar cartões CRC (Class Responsibility Collaboration).
- empregue práticas de meditação (podem ajudar você a focar mais e a ignorar as distrações).

Uma maneira surpreendentemente simples de melhorar a capacidade de reter informações é pegar um bloco de notas e fazer anotações à medida que as informações forem fornecidas, em vez de deixar que elas passem por você.

Isso tem duas finalidades. Em primeiro lugar, o processo o mantém focado e o ajuda a manter a concentração no assunto. É uma ideia básica, porém notavelmente útil. Em segundo lugar, mesmo que você jogue fora as anotações logo em seguida, o estímulo entre os sentidos ajudará você a recapitular os fatos.

PONTO-CHAVE Faça anotações enquanto estiver aprendendo. Mesmo que você vá jogá-las fora.

O seu estado mental afeta a qualidade de seu aprendizado. Estudos mostram que cultivar uma atitude positiva em relação ao aprendizado melhora significativamente a sua capacidade de retenção. Portanto encontre algo para aprender que o envolva. Estresse e falta de sono irão contribuir para uma incapacidade de se concentrar, reduzindo a sua capacidade de aprender.

Os Quatro Estágios da Competência

Você pode aprender informações falsas e acreditar que estão corretas. No melhor caso, pode ser embaraçoso e, no pior, pode ser perigoso. Isso é demonstrado pelos *Quatro Estágios da Competência* (uma classificação postulada nos anos 40 pelo psicólogo Abraham Maslow). Você pode ter:

Incompetência consciente

Você não conhece algo, mas sabe que não conhece. Essa é uma posição relativamente segura para estar. Provavelmente, você não se importa – não é algo que você precise saber. Ou você sabe que ignora o assunto e isso é uma fonte de frustração.

Competência consciente

Também é um bom estado para estar. Você conhece um assunto e sabe que o conhece. Para usar essa habilidade, você deve fazer um esforço consciente e se concentrar.

Competência inconsciente

É quando o seu conhecimento sobre um assunto é tão grande que se torna natural. Você não está mais ciente de que está usando o seu conhecimento. A maioria dos adultos, por exemplo, pode considerar andar e se equilibrar como uma competência inconsciente – simplesmente fazemos isso sem pensar duas vezes.

Incompetência inconsciente

É uma posição perigosa para estar. Você não sabe que não sabe de algo. É ignorante em relação à sua ignorância. De fato, é bem possível que você ache que entenda do assunto, porém não sabe o quanto está errado. É um *ponto cego* em seu conhecimento.

Modelos de aprendizado

Há vários modelos realmente instrutivos de aprendizado, criados por psicólogos da área de educação. O *modelo Dreyfus de aquisição de habilidades é* um exemplo particularmente interessante, postulado pelos irmãos Stuart e Hubert Dreyfus em 1980, quando trabalhavam com inteligência artificial em computadores.[2] Após analisar praticantes altamente habilidosos de certas atividades como pilotos de aviões e grandes mestres do xadrez, eles identificaram cinco níveis específicos de compreensão:

Novato

É um completo iniciante. Os novatos querem ter resultados rápidos, porém não têm nenhuma experiência que os oriente a fazer isso. Eles procuram regras que possam seguir como rotina e não têm discernimento para saber se essas regras são boas ou ruins. Se conseguirem regras boas (ou se recursos adequados forem, por sorte, encontrados no Google), os novatos poderão ir bem longe. Eles *não*

têm nenhum conhecimento sobre um assunto (ainda).

Iniciante avançado

Nesse nível, alguma experiência levou ao aprendizado; você pode se livrar um pouco das regras e tentar realizar tarefas por conta própria. Porém a percepção continua limitada e você não saberá o que fazer quando algo der errado. Nesse nível, há uma melhor compreensão sobre onde obter respostas (você sabe quais são as melhores referências para a API, por exemplo), mas ainda não está em um nível que lhe permita compreender o quadro geral. O iniciante não consegue tirar os detalhes irrelevantes do foco; no que lhe diz respeito, toda e qualquer informação pode ser importante para o problema em questão. Os iniciantes adquirem rapidamente um *conhecimento explícito* – o tipo de conhecimento factual que pode ser facilmente registrado por escrito e articulado.

Competente

Nesse estágio, você tem um modelo mental do domínio do problema; você mapeou a base de conhecimentos, começou a associar as partes e compreende a importância relativa dos diferentes aspectos. Essa visão geral permite abordar problemas desconhecidos e planejar caminhos metódicos para abordar esses problemas em vez de mergulhar de cabeça e esperar que as regras o levem até uma solução. Nesse ponto, você procura novas regras ativamente para formular um plano de ataque e começa a perceber as limitações dessas regras.

É uma boa posição para estar.

Proficiente

As pessoas proficientes vão além da competência. Elas têm um entendimento muito melhor do quadro geral e sentem-se frustradas com as simplificações necessárias aos novatos. Elas podem corrigir erros anteriores e refletir sobre suas experiências para trabalhar melhor no futuro. Nesse ponto, você também pode aprender com as experiências de outras pessoas e assimilá-las em seu corpo de conhecimentos. Pessoas proficientes podem interpretar *máximas* (em oposição a regras simplistas) e aplicá-las a um problema (por exemplo,

elas sabem como e quando aplicar padrões de projeto). É fácil identificar e focar somente nas questões que realmente são importantes, ignorando os detalhes irrelevantes com segurança. Nesse ponto, vemos que a pessoa adquiriu um *conhecimento tácito* significativo – um conhecimento que é difícil de ser transferido por meio de exposição e que é adquirido somente por meio de experiência e de uma compreensão profunda.

Expert

É o topo da árvore de aprendizado. Há bem poucos experts. Eles são as autoridades em um assunto; conhecem-no totalmente e podem usar essa habilidade interligada a outras. Podem ensinar os outros (embora, provavelmente, ensinem melhor aos competentes do que aos novatos, pois se conectam melhor com os primeiros). Os experts têm *intuição*, de modo que, em vez de precisar de regras, eles veem naturalmente uma resposta, mesmo que não possam articular o porquê de ela ser a melhor solução.

Por que o modelo de Dreyfus é interessante? É um modelo revelador para entender em que ponto você está no domínio de um assunto e que ajuda a determinar aonde você deve chegar. Você precisa ser um *expert*? A maioria das pessoas é *competente*, e isso é satisfatório (de fato, uma equipe de experts seria pesada demais e, provavelmente, não seria funcional).

O modelo também ilustra como devemos esperar resolver os problemas em cada estágio do aprendizado. Você está procurando regras simples a serem aplicadas, está reunindo máximas avidamente a partir das quais irá adquirir experiência ou está percebendo intuitivamente as respostas? Quanto do “quadro geral” você vê sobre um assunto?

O modelo de Dreyfus também é de grande utilidade para o trabalho em equipe. Se souber em que ponto um colega está no espectro que varia do novato ao expert, você poderá personalizar melhor a sua interação com ele. Isso o ajudará a saber como trabalhar com outras pessoas – se você deve lhes fornecer algumas regras simples, explicar algumas máximas ou deixar que elas agreguem novas informações à sua compreensão geral.

Observe que o modelo de Dreyfus se aplica *por habilidade*. Você pode ser

um expert em determinado assunto e um completo novato em outro. É natural. E isso deve ser também motivo para que você reconheça a sua humildade – mesmo que você saiba tudo o que há para saber sobre design orientado a comportamento, pode ser que você não saiba nada sobre o framework de teste XYZ. Você deve se sentir empolgado com o fato de que há mais para aprender e aumentar o seu expertise em BDD (Behaviour Driven Development, ou Desenvolvimento orientado a comportamento), ao mesmo tempo que continuará sendo humilde por saber que não é um expert infalível em todos os assuntos! Ninguém gosta de sabe-tudo.

O portfólio de conhecimento

Os Programadores Pragmáticos descrevem uma metáfora vívida e potente sobre o aprendizado – falam de seu *portfólio de conhecimento*.[3] Considere o seu conjunto atual de conhecimentos profissionais como um portfólio de investimentos. Essa metáfora enfatiza maravilhosamente como devemos *administrar* as informações que reunimos, investindo de forma prudente para manter o nosso portfólio atual e trazer novos investimentos para reforçá-lo. Considere quais itens você deve retirar do seu portfólio para criar espaço para outros assuntos.

Esteja ciente do equilíbrio que há entre riscos e compensações para os itens de seu portfólio. Alguns assuntos são de conhecimento comum, porém são um investimento seguro a ser mantido – têm baixo risco, são fáceis de aprender e é certo que serão úteis no futuro. Outros investimentos têm risco mais alto – podem *não* fazer parte da corrente principal de tecnologias e de práticas, portanto estudar esses assuntos pode *não* ser compensador no futuro. Porém, se eles passarem a fazer parte da corrente principal de tecnologias, você fará parte de um conjunto bem pequeno de pessoas com experiência nesse nicho e poderá explorar melhor esse conhecimento. Esses investimentos de mais alto risco em conhecimento podem resultar em dividendos muito mais altos no futuro. Você deve ter uma ampla margem de risco e uma variedade saudável de investimentos em conhecimentos.

PONTO-CHAVE Administre conscientemente o seu portfólio de conhecimentos.

Ensine para aprender

Ensinar é aprender duas vezes.

– Joseph Joubert

Uma das maneiras mais eficientes de aprender algo é *ensinando*; explicar um assunto para outra pessoa consolida o conhecimento em sua mente. Ao precisar explicar algo, você é incentivado a ir mais fundo na questão para realmente entender o assunto. Ensinar força você a revisar o material, reforçando-o em sua memória.

Escreva uma postagem de blog sobre o que você aprender, dê uma palestra, ensine a um amigo ou comece a orientar um colega. Cada uma dessas tarefas irá beneficiar você da mesma maneira que beneficiará a outra pessoa.

Einstein disse que *se você não puder explicar algo de maneira simples, é porque você não entendeu direito*. Ao ler um livro ou ouvir um professor falar, é fácil se enganar achando que "conhece" um assunto. Você ouviu a respeito do tópico, porém não houve nenhum teste para verificar em que ponto está o limite de seu conhecimento. Ensinar força esse limite: será preciso responder a perguntas intrincadas que expandirão o seu conhecimento. Se você não puder responder a uma pergunta, a resposta correta será: *não sei, mas vou procurar saber para você*. Ambos terão aprendido.

PONTO-CHAVE Ensine um assunto para aprendê-lo bem.

Faça para aprender

Eu ouço e esqueço. Eu vejo e lembro. Eu faço e entendo.

– Confúcio

Uma técnica essencial de aprendizado consiste em aprender *fazendo*. Ler livros e artigos, assistir a tutoriais online e participar de conferências sobre programação são atitudes boas. Porém, até que você realmente tente

usar uma habilidade, ela será apenas um conjunto abstrato de conceitos em sua cabeça.

Torne concreto o que for abstrato – mergulhe de cabeça no assunto. Experimente.

O ideal é fazer isso *enquanto* você estiver estudando. Dê início a um projeto de teste e tente usar o conhecimento à medida que adquiri-lo. Ao aprender uma linguagem nova, comece a escrever código com ela imediatamente. Tome os exemplos de código lidos e tente usá-los. Brinque com o código; cometa erros, veja o que funciona e o que não funciona.

> **PONTO-CHAVE** Usar o que você acabou de aprender consolida esse conhecimento em sua memória. Experimente usar exemplos, responda a perguntas e crie projetos de estimação.

Usar as informações é uma maneira certeira de entendê-las. Isso irá gerar novas perguntas que o guiarão durante o seu aprendizado.

O que aprendemos?

Diga-me e eu esquecerei. Mostre-me e eu poderei me lembrar. Envolva-me e eu entenderei.

– Confúcio

Você deve assumir a responsabilidade pelo seu próprio aprendizado. Isso não cabe ao seu empregador, ao sistema educacional público, a um orientador que lhe tenha sido designado ou a qualquer outra pessoa.

Você é responsável pelo seu próprio aprendizado. É importante melhorar suas habilidades continuamente para se tornar um melhor desenvolvedor. E para fazer isso é preciso *aprender* a aprender. Para que seja algo satisfatório, você deve aprender a *amar* o que estiver fazendo.

Aprenda a viver para amar o aprendizado.

Perguntas

1. Qual foi a última vez que você esteve em uma situação que exigisse aprender? Como foi a sua abordagem?

2. Qual foi o seu nível de sucesso?
3. Com que rapidez você aprendeu?
4. Como você poderia ter se saído melhor?
5. Você aprendeu e depois trabalhou ou aprendeu enquanto trabalhava? Qual das opções você acha ser mais eficiente?
6. Qual foi a última vez que você ensinou algo a alguém? Como isso afetou a sua compreensão sobre o assunto?
7. Como você pode encontrar tempo para conhecer assuntos novos se estiver sob pressão para trabalhar e produzir?

Veja também

- *Amor pelas linguagens* (Capítulo 29) – Uma nova linguagem é algo que um programador aprende com frequência.
- *Manifestos* (Capítulo 37) – Conheça as ideias, as tendências, a moda, os modismos e os movimentos mais recentes do mundo do desenvolvimento de software.
- *Desenvolvedores orientados a testes* (Capítulo 25) – Considere como *provar* que você tem uma boa habilidade de programação. Exames e certificações são valiosos?
- *Percorrendo um caminho* (Capítulo 6) – Habilidades aguçadas de aprendizado ajudam a conhecer novas bases de código de forma mais eficiente.

TENTE ISTO... Assuma a responsabilidade pelo seu aprendizado. Tome a iniciativa. Decida o assunto sobre o qual você deve aprender e formule um plano para conhecê-lo.

---

1 É interessante observar que, embora a caracterização referente ao lado esquerdo/direito do cérebro prevaleça em artigos de psicologia popular, nenhum estudo científico provou que uma distinção como essa realmente exista.

2 Stuart E. Dreyfus e Hubert L. Dreyfus, *A Five-Stage Model of the Mental Activities Involved in Directed Skill Acquisition* (Um modelo de cinco estágios das atividades mentais envolvidas na aquisição direcionada de habilidades, Washington, DC: Storming Media).

3 Hunt e Thomas, *The Pragmatic Programmer* (O programador pragmático).

CAPÍTULO 25

Desenvolvedores orientados a testes

A lógica leva você de A para B. A imaginação o leva a qualquer lugar.

– Albert Einstein

Depois de anos preso na produção de software e de muitas horas longas de experiências amargas, o desenvolvimento de software tornou-se natural. Depois que você estiver familiarizado com a sintaxe de sua linguagem de programação, de ter compreendido os conceitos de design dos programas e de ter aprendido a avaliar a diferença entre um código bom e um código ruim, você se verá naturalmente tomando decisões razoáveis de codificação sem muito esforço. Atividades diárias de codificação e "pequenos designs" se tornam instintivos. A sintaxe correta flui da memória dos músculos de seus dedos.

Uma abordagem descuidada, em que "atiramos sem sacar a arma", é sintoma de um "programador cowboy", porém programadores experientes podem trabalhar de maneira extremamente eficiente sem pensar muito. É a vantagem que a experiência traz.

Você já alcançou esse estágio?

De acordo com o modelo de Quatro Estágios da Competência descrito no capítulo 24 (Aprenda a amar o aprendizado), esse estado ideal é o da *competência inconsciente*. É um ato que podemos fazer sem precisar pensar conscientemente; é uma tarefa que podemos executar de maneira eficiente sem nem mesmo perceber exatamente o que estamos fazendo e o seu grau de dificuldade.

Há várias atividades para as quais atingimos um estado de competência inconsciente. Algumas são profissionais. Outras são muito mais mundanas: a maioria dos seres humanos pode andar e comer sem muita

concentração. Uma tarefa comum em que as pessoas veem suas habilidades progredirem, passando pelos quatro estágios da competência, é dirigir um automóvel.

Dirigir oferece uma analogia interessante com a programação. Aprender a dirigir tem muitos paralelos com o aprendizado de nossa arte, e há lições que podemos aprender ao comparar essas duas atividades.

Esclarecendo a questão

Tornar-se um motorista competente exige um aprendizado significativo. É necessário esforço para conhecer o funcionamento do carro bem como as etiquetas e as regras de trânsito. Dirigir bem exige uma coordenação de ações e de habilidades; é um processo intrincado. Você deve investir bastante esforço e prática para atingir a competência.

Quando novos motoristas passam pelo seu primeiro exame de habilitação, eles estão no estágio da *competência consciente* do aprendizado. Eles sabem que podem dirigir e que devem prestar atenção para coordenar cuidadosamente todas as forças em ação. A seleção de uma nova marcha é um processo consciente (para os motoristas iluminados que usam câmbio manual). O domínio da embreagem exige um equilíbrio cuidadoso.

Porém, com a experiência, muitas dessas ações se tornam reações automáticas. Adquirimos autoconfiança. Controlar e lidar com o veículo tornam-se naturais. Nós nos acostumamos ao modo como o veículo responde ao nosso controle. Adotamos naturalmente o posicionamento correto na estrada. Nós nos tornamos mestres da operação do veículo.

Depois que um motorista alcançar esse estágio, sua atenção será liberada para se concentrar no restante que for desconhecido: a própria estrada e as decisões que se apresentam constantemente.

De modo semelhante, depois que os desenvolvedores de software dominam suas ferramentas e linguagens, eles se tornam livres para ver o quadro mais amplo do problema a ser resolvido. Eles podem planejar uma rota sem ter de se concentrar nas minúcias de como farão isso.

Alguns motoristas são melhores que outros. Alguns são mais conscienciosos. Outros têm mais habilidades naturais.

De modo semelhante, alguns desenvolvedores são “naturais”. Outros devem investir bastante esforço para trabalhar de modo eficiente. Alguns desenvolvedores são mais organizados e mais cuidadosos que outros. Alguns não são zelosos e deixam de perceber o que está acontecendo ao seu redor.

A maioria dos problemas na estrada – os acidentes, os atrasos e assim por diante – deve-se a um *erro do motorista*. Os acidentes ocorrem com os carros, porém são provocados por pessoas que aprenderam a usá-los.

A maioria dos desastres de codificação se deve a *erros do programador*. Falhas ocorrem nos programas, porém são provocadas pelas pessoas que aprenderam a escrevê-los.

O sucesso gera complacência

O sucesso gera complacência. A complacência gera falhas. Somente os paranoicos sobrevivem.

– Andy Grove

O estado de competência consciente pode levar à complacência se você não tiver cuidado. Motoristas complacentes não se concentram na estrada e acabam dirigindo no acostamento, sem o devido cuidado e a devida atenção. Em vez de prestar atenção nos perigos na estrada à frente, você pensa no que irá comer no jantar ou canta algo que está tocando no rádio.

Para se tornar um motorista melhor, é importante superar essa complacência. Caso contrário, você será uma ameaça: um perigo real e verdadeiro. Você poderá facilmente colidir com algo ou atropelar alguém.

A programação apresenta armadilhas paralelas. A menos que você tome cuidado, você implementará um código que será uma catástrofe. Lembre-se de que um código descuidado pode custar vidas!

PONTO-CHAVE Tome cuidado para não se tornar complacente ao atingir o estado de “competência”. Sempre codifique com o seu cérebro totalmente engajado

para evitar erros tolos e potencialmente perigosos.

É hora do exame

Antes de sair por aí com um veículo, você deve provar que é capaz. Deve passar pelo *exame de habilitação*. É ilegal dirigir em vias públicas sem antes ter passado nesse exame. O exame de habilitação prova que você tem as habilidades necessárias, além de responsabilidade, para dirigir. Ele demonstra que você não só pode lidar com um carro, mas também pode tomar boas decisões nas condições sob pressão de uma estrada.

A existência de um teste garante que todos os motoristas da estrada atingiram um determinado padrão e que concluíram um determinado volume de treinamento. Esse treinamento indica que:

- Motoristas em aprendizado devem reunir horas de experiência dirigindo no *mundo real* antes de estarem prontos para tentarem o exame. Eles não estudam apenas a teoria sobre direção e entendem o funcionamento de um carro, mas também adquirem experiência prática na estrada. Enquanto estiverem aprendendo, eles são verdadeiros "motoristas aprendizes" estudando para se tornarem mestres.
- Há um risco menor de acidentes nas estradas; os motoristas se informam sobre os perigos e as armadilhas inerentes à direção e aprendem como evitá-los.
- Motoristas treinados e experientes têm mais confiança em suas habilidades e podem tomar decisões maduras.
- Os motoristas compreendem que a estrada é compartilhada com todos os tipos de usuários e levam os outros usuários em consideração.
- Os motoristas estão cientes das limitações de seu equipamento. Eles sabem como reagir em situações de emergência, quando algo der errado.

O exame de habilitação garante que uma atividade humana complexa não acabe em desastre. Ele não só *incentiva* as pessoas a serem bons

motoristas com base em boas intenções, mas *exige* que elas o sejam.

Alguns países vão mais além e têm um exame adicional “avançado” de habilitação – um padrão mais alto para a capacidade de dirigir. Esse exame é uma exigência para determinados empregos.

Desenvolvedores orientados a testes

Não há um equivalente direto de um exame de habilitação no mundo da programação; a certificação não é um pré-requisito legal para escrever códigos (nem deveria ser, na opinião do autor). Contudo, para conseguir um bom emprego, você *realmente* deve demonstrar que tem um nível razoável de habilidades: ter passado em um curso de treinamento de boa reputação ou mostrar que tem uma experiência anterior tangível.

Então aqui está o raciocínio experimental: como seria o equivalente a um exame de habilitação para os desenvolvedores de software? Como você pode demonstrar competência de maneira realista? Faz sentido sequer tentar isso?

Tenho certeza de que há programadores a quem você respeita e *reconhece* como avançados. Porém é possível, prático ou útil certificá-los como tal?

Debatemos sobre o verdadeiro valor de uma certificação em nosso mercado. Certamente, muitas das certificações vendidas pelas empresas de treinamento são pura bobagem; elas ajudam a assinalar itens no formulário de uma solicitação de emprego, porém não querem dizer muito. Você tem uma *certificação de scrum master*? Que maravilha. Espero que não tenha custado muito para comprar esse certificado.

Um teste físico de codificação seria útil? Como ele seria? Como ele deveria ser personalizado para áreas específicas de tecnologia? As várias especializações tornariam a criação desses testes impraticável? Como você avaliaria os engenheiros que não trabalhem predominantemente com código?

Como vimos, a maioria das habilidades dos programadores é adquirida com a experiência reunida no trabalho. Portanto reconhecer a progressão por meio de um modelo *aprendiz-trabalhador-mestre* pode ser mais

adequado para nós. Nem todo programador que trabalhe há muito tempo continua a aprender e a aperfeiçoar suas habilidades; nem todos se tornam mestres artesãos. Tempo de trabalho não é suficiente.

De fato, habilidades avançadas de codificação pode ser ortogonal ao caminho típico de promoção do desenvolvedor. Se você servir fielmente a um trabalho durante *n* anos, sua empresa poderá lhe conceder um aumento e permitir que você suba um degrau a mais na escada corporativa. Contudo isso não significa necessariamente que você seja um programador melhor do que era quando havia iniciado.

Não está totalmente claro que as vantagens de um exame de habilitação poderiam ser trazidas para a nossa profissão de maneira significativa.

Conclusão

Pense por si e deixe que os outros apreciem o privilégio de fazer o mesmo também.

– Voltaire

Essa é uma pequena experiência de raciocínio – uma questão retórica. Nada mais. Porém é interessante pensar sobre esse tipo de assunto e ter um modelo que nos ajude a nos tornar melhores programadores.

Certamente, vale a pena considerar os estágios nas habilidades de codificação pelos quais progredimos. Determine o momento em que você passou da competência consciente para a competência inconsciente. E tome cuidado com a falta de atenção e a complacência.

Perguntas

1. O que você acha? Qual é o equivalente a um exame de habilitação para o programador? Pode haver algo desse tipo?
2. Suas habilidades de programação estão no nível padrão ou no nível avançado? Você acha que, com frequência, você atinge a *competência inconsciente*?
3. Você quer manter o seu nível atual de habilidades? Quer melhorá-lo? Como você fará isso?
4. Como é possível testar a habilidade de um programador em realizar

uma “parada de emergência”?!

5. Há algum valor adicional a ser ganho, resultante do investimento em suas habilidades? Se bons motoristas têm seguros mais baratos como recompensa, como ser um “programador mais seguro” beneficiaria você materialmente?
6. Se codificar é como dirigir, nós tratamos os responsáveis pelos testes de código como bonecos de testes de acidente?

Veja também

- *Aprenda a amar o aprendizado* (Capítulo 24) – Descreve os modelos de aprendizado e o modelo dos Quatro Estágios da Competência em detalhes.
- *Aprecie o desafio* (Capítulo 26) – Independentemente de nosso conhecimento ser formalmente testado, devemos nos esforçar para aperfeiçoar continuamente as nossas habilidades.

> **TENTE ISTO...** Considere a maneira de mudar seus hábitos para se tornar um programador mais atento e menos complacente. Certifique-se de que você não passará da *competência inconsciente* para uma *codificação cowboy*.

CAPÍTULO 26

Aprecie o desafio

O sucesso não é definitivo, a falha não é fatal: é a coragem para continuar que conta.

– Winston Churchill

Somos "trabalhadores do conhecimento". Empregamos nossas habilidades e nossos conhecimentos de tecnologia para que coisas boas aconteçam. Ou para corrigi-las quando isso não ocorrer. É o que nos satisfaz. É para isso que vivemos. Nós nos deleitamos com a oportunidade de criar algo, resolver problemas, trabalhar com novas tecnologias e juntar peças que completem quebra-cabeças interessantes.

Somos programados dessa maneira. Apreciamos o desafio.

O programador envolvido e ativo está constantemente procurando um desafio novo e empolgante.

Dê uma olhada em si mesmo agora. Você procura ativamente novos desafios em sua vida de programação? Vai à caça de novos problemas ou de assuntos em que você realmente esteja interessado? Ou você está simplesmente passando de uma tarefa para a próxima, sem pensar muito sobre o que o motiva?

É possível fazer algo a respeito disso?

É a motivação

Trabalhar com algo estimulante, desafiador ou com que você goste de estar envolvido ajuda a mantê-lo motivado.

Por outro lado, se ficar preso a uma "fábrica de linguiça" de códigos – somente misturando o mesmo velho código por obrigação –, você deixará de prestar atenção. Irá parar de aprender. Irá parar de se importar e de

investir em criar o melhor código que puder. A qualidade de seu trabalho sofrerá as consequências. E sua paixão irá se desvanecer.

Você deixará de melhorar.

Por outro lado, trabalhar ativamente com problemas de codificação que o *desafiem* irá estimulá-lo, empolgá-lo e ajudá-lo a aprender e a se desenvolver. Impedirá que você fique parado e estagnado.

Ninguém gosta de um programador estagnado. Muito menos você.

Qual é o desafio?

Então o que é que particularmente lhe interessa?

Pode ser aquela nova linguagem sobre a qual você tem lido. Ou pode ser trabalhar em uma plataforma diferente. Pode ser simplesmente experimentar um novo algoritmo ou uma nova biblioteca. Ou dar início àquele projeto de estimação em que você pensou há algum tempo. Pode até mesmo ser uma tentativa de otimizar ou de refatorar o seu sistema atual, só para que ele pareça elegante, mesmo que – fico arrepiado só de pensar – isso não resulte em nenhum valor para os negócios.

Com frequência, esse tipo de desafio pessoal só pode ser conseguido em um projeto secundário; algo em que você trabalhe paralelamente às tarefas cotidianas mais mundanas. E isso é *perfeito* – é o antídoto para um desenvolvimento "profissional" enfadonho. Uma panaceia na programação. A cura para um código ruim.

O que o empolga no que diz respeito à programação? Pense em que você gostaria de trabalhar neste momento e por quê:

- Você está satisfeito em ser pago para produzir um código qualquer ou quer ser pago porque faz um trabalho particularmente excepcional?
- Está realizando tarefas pela glória? Você espera ter o reconhecimento de seus colegas ou receber aplausos dos gerentes?
- Quer trabalhar em um projeto de código aberto? Compartilhar o seu código lhe dá uma sensação de satisfação?
- Quer ser a primeira pessoa a fornecer uma solução em um novo nicho

ou para um novo problema complicado?

- Você resolve problemas pela satisfação do exercício intelectual?
- Você gosta de trabalhar em um tipo particular de projeto, ou determinadas tecnologias são mais adequadas às suas tendências estranhas?
- Quer trabalhar junto com determinados tipos de desenvolvedores e aprender com eles?
- Você observa os projetos com olhos empreendedores – procurando algo que ache que algum dia o fará ganhar milhões?

À medida que olho para trás e vejo minha carreira, percebo que tentei trabalhar em várias dessas áreas. Porém me diverti mais e produzi o melhor software quando trabalhei em projetos com os quais eu estava comprometido; quando eu me *importava* com o projeto em si e queria escrever um código excepcional.

Não faça isso!

É claro que há desvantagens em potencial em procurar problemas interessantes de codificação somente "por diversão". Há motivos perfeitamente válidos para não fazer isso:

- Voltar-se para assuntos empolgantes o tempo todo, deixando as partes enfadonhas para os outros programadores assumirem, é ser egoísta.
- É perigoso "fazer ajustes" em um sistema que estiver funcionando somente pelo ajuste em si se isso não estiver agregando nenhum valor verdadeiro aos negócios. Você estará acrescentando alterações e riscos desnecessários. Em um ambiente comercial, é um desperdício de tempo que poderia ser investido em algo mais rentável.
- Se você se distrair com projetos de estimação ou com pequenos "experimentos científicos", jamais terminará nenhum trabalho "de verdade".
- Lembre-se de que nem toda tarefa de programação *será* glamorosa ou empolgante. Muitas de nossas tarefas cotidianas parecem serviços comuns de encanamento. É simplesmente a natureza da programação

no mundo real.

- A vida é curta demais. Não quero desperdiçar meu tempo livre trabalhando com código também!
- Reescrever algo que já existe é um terrível desperdício de esforço. Você não estará contribuindo com o corpo de conhecimentos de nossa profissão. É provável que você vá somente recriar algo que já exista, possivelmente de modo não tão bom quanto as implementações existentes, e que estará cheio de bugs terríveis. Que desperdício de tempo!

Blá blá blá.

Essas posições têm algum mérito. Porém elas não devem se transformar em desculpas que nos impeçam de nos tornarmos melhores programadores.

Exatamente *porque* devemos realizar tarefas enfadonhas o dia todo é que devemos também procurar equilibrá-las com desafios empolgantes. Devemos ser responsáveis em relação ao uso de nosso tempo e saber se usaremos o código resultante ou o jogaremos fora.

Sinta-se desafiado

Descubra o que você gostaria de fazer. E então faça:

- Execute alguns "katas" com o código – pequenos exercícios práticos – que representarão um exercício deliberado e valioso. Depois jogue fora o código.
- Encontre um problema de codificação que você gostaria de resolver somente por diversão.
- Inicie um projeto pessoal. Não desperdice todo o seu tempo livre com ele, mas encontre uma tarefa em que você possa investir esforços e que lhe ensinará algo novo.
- Mantenha um campo amplo de interesse pessoal para que você tenha boas ideias sobre outros assuntos a serem investigados e aprendidos.
- Não ignore outras plataformas e paradigmas. Procure reescrever algo

que você conheça e ame em outra plataforma ou em outro tipo de linguagem de programação. Compare e contraste o resultado. Qual ambiente teve melhor resultado para esse tipo de problema?

- Considere procurar um novo emprego se você não estiver sendo impulsionado e desafiado no local em que estiver trabalhando no momento. Não aceite cegamente o *status quo*! Às vezes, é preciso dar uma chacoalhada.
- Trabalhe ou se reúna com outros programadores motivados. Procure ir a conferências de programação ou junte-se a grupos locais de usuários. Os participantes retornam com a cabeça cheia de novas ideias e revigorados com o entusiasmo de seus colegas.
- Certifique-se de que você possa ver o progresso que está fazendo. Revise os logs do sistema de controle de versões para ver o que foi alcançado. Mantenha um registro diário ou uma lista de tarefas pendentes. Aprecie a eliminação dos itens à medida que avançar.
- Mantenha a cabeça fresca: faça pausas para não se sentir sobrecarregado, sufocado ou entediado com um código.
- Não tenha medo de reinventar a roda! Escreva algo que já tenha sido escrito antes. Não há nenhum mal em tentar escrever o seu próprio componente de lista ligada ou uma GUI padrão. É realmente um bom exercício ver como a sua implementação se compara com as implementações existentes. (Só tenha cuidado em descobrir como irá empregá-la na prática.)

Conclusão

É impraticável e perigoso simplesmente ir atrás somente de assuntos interessantes o tempo todo e deixar de escrever códigos práticos e úteis. Porém é também pessoalmente perigoso ficar preso a uma única trilha de codificação, trabalhando sempre em um software sem sentido e enfadonho, sem ser desafiado ou se divertir.

Você tem algo com o qual está envolvido e em que ama trabalhar?

Perguntas

1. Você tem projetos que o desafiam e que estimulam suas habilidades?
2. Você tem ideias para projetos em que tenha pensado por um tempo, porém não os iniciou? Por que não começar um pequeno projeto secundário?
3. Você equilibra desafios "interessantes" com o seu trabalho cotidiano?
4. Você é desafiado por outros programadores motivados ao seu redor?
5. Você tem um campo de interesse amplo que reforce as bases de seu trabalho?

Veja também

- *Viva para amar o aprendizado* (Capítulo 24) – Quando você estiver entusiasmado para adquirir uma nova habilidade, será preciso empregar técnicas eficientes de aprendizado.
- *Evite a estagnação* (Capítulo 27) – Mantenha a motivação e procure novos desafios para evitar que suas habilidades e a sua carreira fiquem estagnadas.

TENTE ISTO... Considere em que você realmente *gostaria* de trabalhar no momento. É o seu emprego atual? Sorte sua! Caso contrário, como você pode trabalhar com isso agora? Você deve iniciar um "projeto de estimação" ou mudar de emprego?

10.000 MACACOS

(OU ALGO POR AÍ)

DESAFIOS PARA O PROGRAMADOR MODERNO

CAPÍTULO 27

Evite a estagnação

O ferro enferruja por falta de uso; a água perde a sua pureza por ficar estagnada... igualmente, a falta de ação acaba com o vigor da mente.

– Leonardo da Vinci

Quando foi a última vez que você aprendeu algo novo e empolgante o suficiente para colocar em seu currículo? Quando foi a última vez que você foi forçado a ir além de suas capacidades? Quando foi a última vez que o seu trabalho o fez se sentir desconfortável? Quando foi a última vez que você descobriu algo que o deixou encantado? Quando foi a última vez que você se sentiu humilde diante de outro programador e se sentiu estimulado a aprender com ele?

Se as respostas a essas perguntas estiverem em um passado obscuro e distante[1], é porque você entrou na *zona de conforto*: um local que alguns consideram como o nirvana – em que sua vida é fácil e os seus dias de trabalho são curtos e previsíveis.

Entretanto a zona de conforto é um local perigoso. É uma armadilha. Uma vida fácil significa que você não está aprendendo, não está progredindo e não está *melhorando*. A zona de conforto é o local da estagnação. Em breve, você será superado por desenvolvedores mais jovens em ascensão. A zona de conforto é uma via expressa para a obsolescência.

> **PONTO-CHAVE** Tome cuidado com a estagnação. Procurar ser um programador melhor, por definição, não é o estilo de vida dos mais confortáveis.

Poucas pessoas tomam uma decisão consciente de se tornar estagnado. Porém pode ser fácil esgueirar para a zona de conforto e ficar à margem de sua carreira de desenvolvimento sem perceber. Faça uma avaliação de sua

realidade: isso é o que está acontecendo com você nesse momento?

Suas habilidades são o seu investimento

Tome cuidado: preservar o seu conjunto de habilidades é um trabalho árduo. Envolve colocar-se em situações desconfortáveis. Exige um verdadeiro investimento em esforços. Pode ser arriscado e difícil. Você pode até mesmo passar vexame. Isso não soa totalmente agradável, não é mesmo?

Sendo assim, não é algo que muitas pessoas se sintam naturalmente inclinadas a fazer. Você gasta muitas horas do dia trabalhando; você não mereceria ter uma vida fácil e então ir para casa e se esquecer de tudo? É natural inclinar-se em direção ao familiar e ao confortável.

Não faça isso!

Você deve tomar uma decisão consciente de investir em suas habilidades. E deve tomar essa decisão repetidamente. Não a veja como uma tarefa árdua. Deleite-se com o desafio. Aprecie o fato de estar fazendo um investimento que o tornará um programador melhor, além de uma pessoa melhor.

> **PONTO-CHAVE** Espere investir tempo e esforço para ampliar o seu conjunto de habilidades. É um investimento que vale a pena fazer; será compensador.

Um exercício para o leitor

Como você pode dar uma chacoalhada em si mesmo nesse momento? A seguir, temos algumas mudanças que podem ser feitas para tirá-lo da zona de conforto:

- Pare de usar as mesmas ferramentas; podem haver ferramentas melhores que tornarão sua vida mais fácil se você as conhecesse.
- Pare de usar a mesma linguagem de programação para todos os problemas; pode ser que você esteja quebrando uma noz com uma marreta.
- Comece a usar um sistema operacional diferente. Aprenda a usá-lo

apropriadamente. Mesmo que seja um sistema de que você não goste, passe um tempo experimentando-o para realmente conhecer os seus pontos fortes e os pontos fracos.

- Comece a usar um editor de texto diferente.
- Aprenda a usar atalhos de teclado e veja como isso impacta o seu fluxo de trabalho. Faça um esforço consciente para parar de usar o mouse.
- Conheça um assunto novo: algo que, no momento, você não *precise* conhecer. Aprofunde, talvez, o seu conhecimento de matemática ou de algoritmos de ordenação.
- Inicie um projeto pessoal de estimação. Sim, use parte de seu precioso tempo livre para ser um geek. Publique-o como um projeto de código aberto.
- Direcione o seu trabalho para uma nova parte de seu projeto – uma parte que você conheça pouco. Talvez você não seja produtivo de imediato, porém terá um conhecimento mais amplo do código e aprenderá assuntos novos.

Considere expandir-se além do campo da programação:

- Aprenda uma nova linguagem. Mas *não* uma linguagem de programação. Ouça audiolivros que ensinem a contar em japonês enquanto dirige para o trabalho.
- Reorganize a sua escrivaninha! Tente ver a maneira como você trabalha a partir de outro ângulo.
- Comece uma nova atividade. Talvez iniciar um blog para registrar o seu aprendizado em um diário. Invista mais tempo em um hobby.
- Comece a se exercitar: inscreva-se em uma academia ou comece a correr.
- Socialize-se mais. Passe tempo com geeks *e* com não geeks.
- Considere fazer ajustes em sua dieta. Ou vá dormir mais cedo.

Segurança no emprego

Ser um desenvolvedor melhor – que tenha um conjunto mais afiado de

habilidades e que esteja constantemente aprendendo – dará uma sensação de mais segurança em seu emprego. Porém pergunte a si mesmo se você realmente precisa disso: você *está* no emprego certo?

Esperamos que você esteja na carreira certa: você gosta de programar. (Se não gostar, considere seriamente se uma mudança de carreira não seria uma boa opção. O que você *realmente* gostaria de fazer?)

Há um perigo em permanecer em um emprego ou em uma função por tempo demais, em fazer o mesmo trabalho repetidamente, sem que haja novos desafios. Com muita facilidade, acabamos imersos naquilo que estamos fazendo. Gostamos de ser experts locais – o rei de nosso pequeno castelo de codificação. É confortável.

Quem sabe agora não seja a hora de mudar de empresa? De encarar novos desafios e prosseguir em sua jornada de codificação. De escapar da zona de conforto.

Permanecer no mesmo local geralmente é mais fácil, mais familiar e mais conveniente. No recente clima econômico efervescente, também é uma aposta mais segura. Contudo pode não ser a melhor opção para você. Um bom programador é corajoso tanto em sua abordagem em relação ao código quanto em relação a sua carreira.

Perguntas

1. Você está estagnado no momento? Como você pode afirmar isso?
2. Qual foi a última novidade que você aprendeu?
3. Qual foi a última vez que você aprendeu uma nova linguagem? Qual foi a última vez que você aprendeu uma técnica nova?
4. Que nova habilidade você deverá adquirir em seguida? Como você irá adquiri-la? Quais livros, cursos ou material online você usará?
5. Você está no emprego certo nesse momento? Você gosta dele ou toda a alegria já se esvaiu? Você está trabalhando somente para cumprir o horário obrigatório ou está entusiasmado e engajado para ver o projeto ser bem-sucedido? Você deve procurar novos desafios?
6. Qual foi a última vez que você recebeu uma promoção? Ou um

aumento de salário? Um nome de cargo tem um significado real? O nome de seu cargo tem alguma relação com suas habilidades?

Veja também

- *Aprecie o desafio* (Capítulo 26) – Evite a estagnação procurando novos desafios.

TENTE ISTO... Comprometa-se agora para evitar a estagnação. Faça uma avaliação honesta para saber até que ponto você está "preso na lama" nesse momento. Defina um plano prático para melhorar.

---

1 O "passado distante" não é tão distante assim quando medido em *anos de programador*, motivo pelo qual as pessoas acham tão difícil estimar a duração de projetos de software!

CAPÍTULO 28

O programador ético

Eu poderia muito bem responder-lhe dizendo: "Estás errado, meu amigo, se achas que um homem de algum valor deva gastar seu tempo avaliando os prospectos sobre a vida e a morte. Ele tem apenas um aspecto a considerar ao realizar qualquer ação – é saber se está agindo de maneira certa ou errada, como um homem bom ou um homem mau."

– Sócrates
Apologia

Com frequência, descrevo como a qualidade de um programador depende mais de sua *atitude* do que de sua maestria técnica. Uma conversa recente sobre esse assunto me levou a considerar a questão sobre o *programador ético*.

O que isso quer dizer? Com que isso se parece? A ética tem alguma parte a ser considerada na vida do programador?

É impossível separar o ato da programação de qualquer outra parte da existência humana do programador. Desse modo, naturalmente, as preocupações éticas governam o que nós, como programadores, fazemos e como nos relacionamos profissionalmente com as pessoas.

É lógico pensar então que ser um "programador ético" vale a pena; ao menos, vale tanto a pena quanto ser uma *pessoa* ética. Certamente, você se preocuparia com qualquer um que aspirasse a ser um *programador antiético*.

Muitas profissões têm códigos específicos de ética para a sua conduta. A medicina tem o juramento de Hipócrates, que compromete os médicos a trabalhar para o bem de seus pacientes e a não prejudicá-los. Advogados e engenheiros têm suas próprias agremiações profissionais que lhes conferem um *status* como associados, exigindo que seus membros estejam de acordo com determinadas regras de conduta. Esses códigos de ética

existem para proteger seus clientes, resguardar seus profissionais e também para garantir o bom nome da profissão.

Em "engenharia" de software, não temos regras universais como essas. Há poucos padrões de mercado que possam ser atribuídos a nós de maneira útil. Várias organizações publicam seu próprio código de ética, por exemplo, o ACM (*http://www.acm.org/about/code-of-ethics*) e o BSI (*http://www.bcs.org/server.php?show=nav.6030*). Entretanto eles têm pouca representatividade do ponto de vista legal e não são universalmente reconhecidos.

A ética em nosso trabalho é amplamente determinada por nossas próprias atitudes morais. Certamente, há vários excelentes programadores por aí que trabalham por amor a sua arte e pelo progresso da profissão. Há também outros tipos mais obscuros que estão no jogo predominantemente em causa própria. Já conheci os dois tipos.

A ética computacional inicialmente foi um termo cunhado por Walter Maner em meados da década de 70. Assim como outros assuntos relacionados ao estudo da ética, esse é considerado um ramo da filosofia.

Trabalhar como um programador "ético" tem aspectos a serem considerados em diversas áreas: notadamente em nossas atitudes em relação ao código e às pessoas. Também há várias questões legais que devem ser compreendidas. Daremos uma olhada nelas nas seções a seguir.

Atitude em relação ao código

Não escreva um código que seja propositalmente difícil de ler ou que tenha um design complexo a ponto de ninguém ser capaz de segui-lo.

Fazemos piada sobre isso ser um esquema para "garantir o emprego": escrever códigos que somente você possa ler garante que você jamais será demitido! Programadores éticos sabem que a garantia de seus empregos está em seu talento, na integridade e em seu valor para uma empresa, e não em sua capacidade de fazer com que a empresa dependa deles.

PONTO-CHAVE Não se torne "indispensável" escrevendo um código ilegível ou desnecessariamente "inteligente".

Não “corrija” bugs usando soluções alternativas com esparadrapos ou fazendo remendos rápidos, ocultando um problema, porém deixando a porta aberta para outras variantes se manifestarem. O programador ético encontra o bug, entende-o e aplica uma correção apropriada, sólida e testada. É a maneira “profissional” de trabalhar.

O que aconteceria se você estivesse a um passo de um prazo inalterável se esgotar e simplesmente fosse *necessário* disponibilizar o código e descobrisse um bug terrível, embaraçoso e que fará o show parar? Seria ético aplicar uma correção rápida temporária para salvar a versão iminente? Talvez. Nesse caso, pode ser uma solução perfeitamente pragmática. Porém o programador ético não deixa a situação como está: ele adiciona uma nova tarefa à lista para monitorar o “débito técnico” contraído e tenta pagá-lo imediatamente após o software ter sido disponibilizado. Esses tipos de solução com esparadrapo não devem ser deixados no código para que o contaminem além do tempo necessário.

O programador ético procura escrever o melhor código possível. Em qualquer momento, trabalhe da melhor maneira que suas habilidades permitirem. Empregue as ferramentas e técnicas mais apropriadas que conduzirão aos melhores resultados – por exemplo, use testes automatizados para garantir a qualidade, faça programação aos pares e/ou revisão de código para identificar os erros e melhorar os designs.

Questões legais

Um profissional ético e profissional compreende a pertinência das questões legais e garante que as regras sejam obedecidas. Considere, por exemplo, o campo espinhoso da licença de software.

Não use código protegido por direitos autorais, como código-fonte GPL (*http://www.gnu.org/licenses/gpl.html*), em códigos proprietários, quando a licença não o permitir.

PONTO-CHAVE Honre as licenças de software.

Ao mudar de emprego, não leve códigos-fontes nem tecnologias da empresa antiga, nem transplante partes deles para uma nova empresa.

Não mostre nem mesmo partes desse código em uma entrevista em outra empresa.

Esse é um assunto interessante, pois conduz a uma área cinzenta maior: copiar propriedade intelectual privada ou um código que tenha uma informação clara de direitos autorais claramente é o mesmo que roubar. Entretanto contratamos os programadores de acordo com suas experiências anteriores – as tarefas que realizaram no passado. Reescrever o mesmo tipo de código de cabeça, sem duplicar exatamente as linhas de código, é ético? Reimplementar outra versão de um algoritmo proprietário que confira uma vantagem competitiva será antiético se você tiver contratado o designer desse algoritmo especificamente devido à sua experiência?

Geralmente, um código é publicado online com uma licença bem liberal, simplesmente solicitando que uma atribuição seja feita. O programador ético toma o cuidado de garantir que essa atribuição seja adequadamente feita em um código desse tipo.

PONTO-CHAVE Garanta que os créditos apropriados sejam conferidos ao trabalho que você reutilizar em sua base de código.

Se souber que há problemas legais em torno de alguma tecnologia que estiver sendo usada (por exemplo, algoritmos de criptografia ou de descriptografia sobre os quais pesem restrições comerciais), você deverá garantir que o seu trabalho não violará essas leis.

Não roube softwares nem use ferramentas de desenvolvimento piratas. Se você receber uma cópia de um IDE, certifique-se de que haja uma licença válida de uso para você. Assim como você não iria criar uma cópia pirata de um filme nem compartilhar músicas online que tenham direitos autorais, não utilize livros técnicos ilegalmente copiados.

Não invada computadores nem dados armazenados para os quais você não tenha autorização de acesso. Se você perceber que é possível acessar um sistema desse tipo, deixe que os administradores saibam para que eles possam corrigir as permissões.

Atitude em relação às pessoas

Trate as outras pessoas como você gostaria que elas o tratassem.

– Mateus 7:12

Já consideramos algumas "atitudes éticas" em relação às pessoas, pois escrevemos códigos principalmente para outros programadores, e não para o compilador. Os problemas de programação quase sempre são problemas de pessoas, mesmo que as soluções tenham uma natureza técnica.

PONTO-CHAVE Boas atitudes em relação ao código *também* são boas atitudes em relação a outros programadores.

Pense em você mesmo como um herói programador. O tipo de programador que vista a roupa íntima sobre as calças, e não simplesmente um geek com roupas cafonas. *Não abuse de seus superpoderes para fazer o mal.* Crie softwares somente para o bem da humanidade.

Na prática, isso quer dizer: não crie vírus nem malwares. Não crie softwares que infrinjam a lei. Não crie softwares para piorar a vida das pessoas, seja materialmente, fisicamente, emocionalmente ou psicologicamente.

Não vá para o lado negro da força.

PONTO-CHAVE Não crie softwares que piorem a vida de outra pessoa. Isso é um abuso de poder.

E, nesse ponto, entramos em um novo balaio de gatos: é ético criar softwares que tornem algumas pessoas muito ricas à custa de pessoas mais pobres se nenhuma lei for infringida? É ético criar softwares para distribuir conteúdo pornográfico se o software em si não infringir nenhuma lei? Você pode argumentar que as pessoas são exploradas como subproduto de ambas as atividades. É ético trabalhar nesses mercados? É uma pergunta que posso deixar somente para o leitor responder.

E trabalhar em projetos militares? Um programador ético se sentiria à vontade trabalhando em sistemas de armamento que poderão ser usados para acabar com uma vida? Talvez um sistema como esse, na realidade,

salve vidas se for usado para deter ataques. Esse é um ótimo exemplo de como a ética no desenvolvimento de software é um assunto filosófico, e não uma questão totalmente do tipo “preto no branco”. Você deve conciliar a si mesmo com as consequências provocadas pelo seu código na vida de outras pessoas.

Colegas de equipe

As pessoas com as quais você mais convive em sua carreira de programação são os seus colegas de equipe – os programadores, os responsáveis pelos testes e assim por diante, que trabalham bem próximos de você no dia a dia. O programador ético trabalha de forma consciente com todos eles, procurando honrar cada membro da equipe e trabalhar em conjunto para atingir os melhores resultados possíveis.

Fale bem de todas as pessoas. Não se envolva com fofocas ou maledicências. Não incentive piadas à custa de outras pessoas.

Sempre acredite que qualquer pessoa, independentemente de seu nível de maturidade ou de experiência, tem algo de valor com que contribuir. Elas têm uma opinião que vale a pena ser ouvida e devem ter o direito de apresentar pontos de vista sem serem menosprezadas.

Seja honesto e confiável. Trate todas as pessoas com integridade.

Não finja concordar com alguém se achar que essa pessoa está errada; isso é desonesto e, raramente, será útil. Discordâncias construtivas e discussões sensatas podem resultar em decisões de design genuinamente melhores para o código. Saiba qual é o nível de “debate” com que um membro da equipe pode lidar. Algumas pessoas se dão bem em debates intelectuais intensos e apaixonados; outras têm medo do confronto. O programador ético procura obter o resultado mais produtivo nas discussões, sem insultar nem ofender ninguém. Isso nem sempre é possível, porém o objetivo é sempre tratar as pessoas com respeito.

Não discrimine ninguém, sob qualquer pretexto, incluindo gênero, raça, capacidade física, orientação sexual, capacidade mental ou habilidade.

O programador ético toma bastante cuidado ao lidar com “pessoas difíceis” da maneira mais justa e transparente possível, tenta atenuar

situações complicadas e trabalha no sentido de evitar conflitos desnecessários.

PONTO-CHAVE Trate as outras pessoas como você gostaria que elas o tratassem.

Gerente

Muitas questões que possam ser vistas como acordos entre você e o seu gerente também podem ser vistas como contratos éticos entre você e os outros membros da equipe, pois o gerente atua como uma ponte para a equipe.

Não aceite créditos por trabalhos que não sejam seus, mesmo que você tenha usado a ideia de outra pessoa e a tenha modificado para enquadrá-la melhor no contexto. Dê crédito a quem se deve.

Não faça uma estimativa desnecessariamente alta para a complexidade de uma tarefa somente para que você possa ser desleixado e fazer algo mais agradável, sob o pretexto de estar trabalhando arduamente em um problema complicado.

Se você vir problemas se assomando, que irão impedir um andamento harmonioso do projeto, relate-os assim que perceber. Não esconda as notícias ruins porque não quer preocupar ou ofender ninguém, nem ser visto como um desmancha-prazeres pessimista. Quanto mais cedo os problemas forem levantados, mais planos forem criados para solucioná-los e os problemas forem tratados, mais tranquilo será o projeto para todos.

Se você identificar um bug no sistema, informe-o. Insira um bug no sistema de rastreamento de falhas. Não finja que é cego, esperando que ninguém mais perceba o problema.

PONTO-CHAVE O programador ético assume a responsabilidade pela qualidade do produto sempre.

Não finja ter habilidades ou conhecimentos técnicos que você não tenha. Não coloque o cronograma de um projeto em perigo comprometendo-se com uma tarefa que você não possa terminar somente porque acha que ela seja interessante e porque gostaria de trabalhar nela.

Se perceber que uma tarefa que você estiver executando vá consumir significativamente muito mais tempo do que você esperava para concluí-la, explicite essa preocupação o mais rápido possível. O programador ético não fica em silêncio somente para manter a compostura.

Ao assumir a responsabilidade por algo, honre a confiança depositada em você. Trabalhe da melhor maneira que suas habilidades permitirem para cumprir essa responsabilidade.

Empregador

Trate o seu empregador com respeito.

Não revele informações confidenciais da empresa, incluindo códigos-fontes, algoritmos ou informações internas. Não infrinja os termos de seu contrato de emprego.

Não venda o trabalho feito em uma empresa para outra, a menos que você tenha permissão expressa para fazê-lo.

No entanto, se você perceber que o seu empregador está envolvido em atividades ilegais, é sua obrigação ética levantar a questão junto a ele ou delatar a sua conduta ilegal, conforme for apropriado. O programador ético não finge ser cego diante de erros somente para garantir o seu próprio emprego.

Não represente falsamente o seu empregador nem fale mal dele publicamente.

Você

Como um programador ético, você deve se manter atualizado em relação às práticas de boa programação.

Os programadores éticos não trabalham tão arduamente a ponto de ficarem esgotados. Isso não é somente desvantajoso pessoalmente, mas também é ruim para toda a equipe. Horas e horas de trabalho extra, semana após semana, resultarão em um programador cansado que, inevitavelmente, cometerá erros por descuido, e o resultado final será pior. O programador ético entende que, embora seja bom parecer um herói que

trabalhe de maneira extremamente árdua, não é uma boa ideia definir expectativas não realistas e ficar esgotado.

> **PONTO-CHAVE** Um programador cansado não é útil a ninguém. Não trabalhe demais. Conheça os seus limites.

Você tem o direito de esperar a mesma conduta ética das outras pessoas com quem você trabalha.

O juramento de Hipocódigo

Como seria o *código de ética* ideal para os programadores? Os documentos do ACM e do BSI para ética são formais, extensos e difíceis de lembrar.

Precisamos de algo mais expressivo; algo mais semelhante a uma definição de missão para o programador ético.

Humildemente, sugiro o seguinte:

> *Juro não causar nenhum dano ao código ou aos negócios; juro procurar fazer progressos pessoais e em minha arte. Realizarei as tarefas a mim alocadas da melhor maneira permitida pelas minhas habilidades, trabalhando harmoniosamente com a minha equipe. Vou me relacionar com as outras pessoas com integridade, trabalhando para maximizar a eficiência e o valor do meu projeto e da minha equipe.*

Conclusão

A ética, em princípio, é a arte de recomendar aos outros os sacrifícios necessários para cooperar com alguém.

– Bertrand Russell

Até que ponto você se importa com esse tipo de assunto depende do seu nível de zelo, de seu profissionalismo e de seu próprio código moral. Você está no jogo da programação por diversão, pelo prazer e para o desenvolvimento de um bom código? Ou está nele em causa própria, pelo desenvolvimento de sua carreira (à custa dos outros, se necessário), para ganhar o máximo de dinheiro que puder e se elevar acima dos outros na escala profissional?

É uma escolha. Você pode escolher a sua atitude. Ela irá moldar a trajetória de sua carreira.

Acho que minhas atitudes são moldadas pelo meu desejo de escrever um bom código e de participar de uma comunidade que se importe em trabalhar bem. Procuro trabalhar com desenvolvedores excelentes, com quem eu possa aprender. Como cristão, tenho um modelo moral que me incentiva a dar preferência aos outros em relação a mim mesmo e a honrar os meus empregadores. Isso molda a minha maneira de agir.

Concluo, de acordo com o que vimos aqui, que há (pelo menos) dois níveis na carreira de um programador ético: "não prejudicar" corresponde ao nível básico, ou seja, não pisar nas pessoas nem se envolver com trabalhos que as explorem. Para além disso, há um mantra ético mais elaborado: trabalhar somente em projetos que resultem em benefícios sociais sólidos, especificamente *tornar o mundo melhor* com os seus talentos e compartilhar conhecimentos para o progresso da arte da programação. Suspeito que muitas pessoas pertençam ao primeiro campo da ética. Poucas sentem a necessidade ou são capazes de se dedicar à causa do segundo.

De que forma suas crenças e atitudes moldam a maneira como você trabalha? Você se considera um programador ético?

Perguntas

1. Você se considera um programador "ético"? Há uma diferença entre ser um programador ético e uma pessoa ética?
2. Você concorda com alguma das observações deste capítulo ou discorda de alguma? Por quê?
3. É ético criar um software que deixe os banqueiros fabulosamente ricos se o dinheiro que eles ganharem for à custa de outras pessoas que não sejam capazes de explorar a mesma capacidade computacional? Faz diferença se a prática comercial é legal ou não?
4. Se a sua empresa estiver usando um código GPL em seus produtos proprietários, porém não estiver cumprindo as obrigações dos termos da licença (retendo o seu próprio código), o que você deve fazer? Você

deve lutar para que os termos da licença sejam atendidos abrindo o código-fonte da empresa? Ou deve pedir que esse código-fonte GPL seja substituído por uma alternativa de código fechado? Se o produto já tiver sido lançado, você deve ser um dedo-duro e expor a violação de licença? E se o seu emprego dependesse de se manter calado?

5. O que você deveria fazer se identificasse outro programador agindo de modo "antiético"? Qual seria a diferença na resposta se esse programador fosse um colega de trabalho, um amigo, alguém a quem você foi solicitado a fornecer referências ou um programador que você conheceu, mas com quem não trabalha diretamente?
6. Como as patentes de software se encaixam no mundo da programação ética?
7. Sua paixão pelo desenvolvimento de software tem alguma influência na importância que você dá às questões éticas? Um programador apaixonado age de modo mais ético do que um programador que só pensa em fazer carreira?

Veja também

- *Ser responsável* (Capítulo 35) – Ser responsável diante de outro programador irá motivar você a agir virtuosamente.
- *O poder das pessoas* (Capítulo 34) – Trate as pessoas com quem você trabalha com respeito. Aprenda com elas; incentive-as a melhorar.

TENTE ISTO... Analise as questões mencionadas neste capítulo. Como você pode garantir que está agindo de forma ética em cada área? Qual área faz você se sentir menos confortável? Dê um passo específico para lidar com isso.

10.000 MACACOS
(OU ALGO POR AÍ)

OS SEGREDOS DA PROGRAMAÇÃO ÉTICA

USAMOS SOMENTE CÓDIGO RECICLADO

CHAMAMOS ISSO DE ESTRATÉGIA DO "DINHEIRO POR CORDA VELHA"

USAMOS CÓDIGOS SOMENTE DE BASES ADMINISTRADAS DE FORMA SUSTENTÁVEL

PARA CADA ÁRVORE BALANCEADA QUE USAMOS, PLANTAMOS MAIS DUAS: UMA VERMELHA E UMA NEGRA

EMPREGAMOS OS MELHORES PROGRAMADORES CRIADOS LIVREMENTE

(ELES ESCREVEM UM CÓDIGO FRANGÁSTICO)

CAPÍTULO 29

Amor pelas linguagens

Aqueles que não conhecem nenhuma língua estrangeira não sabem nada de seu próprio idioma.

– Johann Wolfgang von Goethe
Máximas e reflexões

Não há dois problemas iguais. Dois desafios não são idênticos. E, do mesmo modo, dois programas não são exatamente iguais. Felizmente, isso torna o nosso trabalho interessante.

Porém algumas tarefas *são* semelhantes de forma suspeita. Para nós, é aí que está o dinheiro fácil – reutilizar as habilidades que já adquirimos. Isso é experiência, ou seja, o que torna você valioso no mercado de trabalho. Mas é também o que o torna um desenvolvedor estagnado – um cavalinho de circo que conhece apenas um truque. Um cachorrinho que não conhece nenhum truque novo.

Devemos encarar novos desafios continuamente, aprender continuamente, resolver novos problemas continuamente e usar novas tecnologias continuamente.

É assim que você se torna um programador melhor.

> **PONTO-CHAVE** Não se transforme em um cavalinho de circo que conhece *apenas um truque*. Coloque-se em uma posição de modo a encarar novos desafios, aprender e crescer como desenvolvedor.

Ame todas as linguagens

Parte desse regime de crescimento consiste em trabalhar com mais de uma linguagem. Ficar preso a uma única linguagem fará com que sua resolução de problema seja unidimensional. Muitos desenvolvedores

cultivam uma carreira conhecendo apenas um assunto e perdem um mundo de oportunidades.

Aprenda diversas linguagens para ser fluente em vários tipos de solução. Conheça as linguagens de scripting e as linguagens compiladas. Conheça as linguagens simples, com um mínimo de recursos, e as linguagens com bibliotecas vastas e abrangentes. Acima de tudo, conheça linguagens que sigam idioms e paradigmas diferentes.

Hoje em dia, o trabalho mundano continua sendo feito com *linguagens imperativas* (geralmente, linguagens procedurais)[1], na maioria das vezes orientadas a objetos: C#, Java, C++, Python e as de sua família. O Smalltalk é uma linguagem orientada a objetos interessante, com algumas ideias diferentes que valem a pena conhecer; o design do Objective-C foi diretamente baseado nela. Há linguagens procedurais não orientadas a objetos por aí, embora a maioria tenha partes orientadas a objetos

incluídas atualmente, por exemplo, Basic ou Pascal. A maioria das linguagens de shell scripting continua sendo uma engenhoca puramente procedural.

As *linguagens funcionais* representam um campo particularmente rico para aprender. Mesmo que você não use uma diariamente, entender os conceitos presentes nas linguagens funcionais dará mais base aos programadores procedurais e os ajudarão a escrever um código mais sólido, expressivo e robusto. Vale a pena estudar Lisp, Scheme, Scala, Clojure, Haskell e Erlang.

As *linguagens lógicas*, como o Prolog, ensinam uma maneira totalmente diferente de pensar sobre uma solução e expressá-la. As *linguagens de especificação formal*, como o Z, embora raramente tenham um uso ativo, ilustram até que ponto podemos ser rigorosos. Esse rigor foi transferido para uma aplicação mais prática na linguagem Eiffel, que inclui a noção de contratos fortes em uma linguagem orientada a objetos.

Considere também aprender algumas linguagens "mortas", que não são mais comumente utilizadas, para entender a história de sua arte. O BCPL foi o precursor da família de linguagens do tipo C que usam chaves. Os conceitos do Simula foram usados como base para o design do C++. O COBOL – uma linguagem realmente veterana – historicamente era usado em aplicações de negócios. Muitos desses sistemas continuam em execução atualmente. (Programadores de COBOL ganharam uma fortuna corrigindo o bug do milênio[2]).

Adquira conhecimentos sobre a linguagem assembly: fale (quase) diretamente com a CPU. A maioria dos programadores raramente precisa recorrer a opcodes e mnemônicos enigmáticos, porém entender sobre o que suas linguagens de alto nível estão construídas ajuda a escrever um software melhor.

> **PONTO-CHAVE** Um bom programador conhece muitas linguagens e diversos idioms, ampliando sua paleta de soluções. Isso melhora o código que escrevem.

Embora conhecer várias linguagens seja louvável, é difícil ser extremamente proficiente em todas elas. É provável que você vá focar a

maior parte de seu talento em umas duas delas; do contrário, você correrá o risco de se tornar um *pau para toda obra, porém mestre em nada*.

Ame sua linguagem

No dia a dia, eu uso C++ mais do que qualquer outra linguagem. Portanto irei usá-la como exemplo. Muitas pessoas menosprezam o C++ como sendo excessivamente complexo e arcaico. Não é. Ele é eficaz, expressivo e tem bom desempenho. Porém ele permite dar um tiro no seu próprio pé se você quiser.

Ter usado o C++ furiosamente durante vários anos (às vezes, literalmente em fúria) tem sido uma grande diversão, além de ter sido uma experiência muito esclarecedora. Desse modo, atualmente tenho uma relação de amor e ódio com a linguagem.

Ela não é perfeita. O C++ é uma ferramenta bem afiada e, como todas as ferramentas afiadas, tem pontas afiadas. Com frequência, você pode levar a linguagem a gerar um código incrível e expressivo de maneira que nenhuma outra linguagem consegue fazer. Porém, outras vezes, você pode se cortar acidentalmente com uma de suas pontas afiadas e acabar encarando páginas de erros inexplicáveis de template, murmurando palavrões entre os dentes.

Esses palavrões murmurados não provam que o C++ seja uma linguagem ruim (embora alguns possam argumentar que sim). Toda linguagem tem seus defeitos. É somente uma linguagem com a qual você deve aprender a conviver. Uma linguagem que você deve entender bem para trabalhar com ela de forma adequada. Você deve *pegar o jeito* de como ela funciona internamente e saber o que a faz se movimentar. Ela é mais exigente do que outras opções.

Eu estava pensando sobre isso e, de repente, me ocorreu que...

Estamos muito envolvidos em um verdadeiro *relacionamento* com a nossa linguagem. Pode até ser parecido com um casamento; é um relacionamento gratificante, porém exige trabalho.

PONTO-CHAVE Trabalhar com a sua linguagem de programação é um

relacionamento ao qual você deve se dedicar todos os dias.

Tome cuidado com o modo como você trata a linguagem C++. Ela é uma fera volúvel: oferece várias técnicas maravilhosas e empolgantes para trabalhar e, ao fazer isso, fornece corda suficiente para você se enforcar. Com os templates, há mensagens de erro incompreensíveis relacionadas a tipos. Com a herança múltipla, há um enorme potencial para design ruim de classes e o *problema do diamante* (deadly diamond)[3]. Com `new` e `delete`, temos memory leaks e ponteiros soltos (dangling pointers).

Isso significa que o C++ é *ruim*? Devemos jogar fora o C++ agora e usar Java ou C# em seu lugar? É claro que não! Alguns programadores defendem que sim, porém essa é uma visão errônea e míope. Nada na vida vem de graça; você deve esperar fazer um investimento alto para ter um relacionamento satisfatório. É difícil viver com alguém (ou algo) dia após dia, compartilhar seus pensamentos (programação) mais íntimos e não ficar ocasionalmente chateado com essa pessoa (linguagem). Você pode esperar uma certa quantidade de atrito à medida que se tornar mais íntimo e se acostumar um com o outro. Qualquer relacionamento conduz você a um caminho de constante aprendizado um sobre o outro, de descobrir como acomodar as manias um do outro e como fazer surgir o que há de melhor em cada um.

Devo admitir que o C++ não investirá muitos esforços em conhecer você, mas muitas pessoas inteligentes (o pessoal que projetou e padronizou a fera) já o fizeram.

Tornar-se proficiente em qualquer linguagem exige compromisso. Muitos programadores são avessos a investir o esforço necessário ou ficam frustrados muito facilmente quando algo dá errado. Eles então se prostituem com outras linguagens que acham ser mais satisfatórias, ou partem para uma alternativa mais jovem e glamorosa. (É sempre bom para inflar o ego ser visto com uma linguagem jovem, como se fosse um troféu, em seus braços. Isso seria o equivalente à crise de meia-idade para os programadores?[4])

Cultivando o relacionamento com a sua linguagem

Há algumas marcas registradas geralmente aceitas para um casamento saudável. Elas podem lançar um pouco de luz sobre um relacionamento saudável com a sua linguagem.

Um bom casamento exige amor e respeito, compromisso, comunicação, paciência e valores compartilhados. Vamos dar uma olhada com mais detalhes.

Amor e respeito

Para um casamento ser bem-sucedido, os parceiros devem gostar um do outro, devem valorizar um ao outro ou querer passar tempo um com o outro. Deve haver um nível de atração. Eles devem amar um ao outro.

A maioria dos programadores programa porque é sua paixão. Eles amam escrever código. E normalmente eles escolhem uma linguagem porque realmente gostam de usá-la.

> **PONTO-CHAVE** Ame a sua linguagem! Trabalhe com uma linguagem que você aprecie.

No entanto muitas pessoas são forçadas a usar uma determinada linguagem no trabalho porque a base de código existente está escrita nela – nesse sentido, elas entram em um casamento arranjado. Em suas casas, essas pessoas prefeririam lidar com uma porção interessante de Ruby ou de Python. Lembre-se de que alguns casamentos arranjados funcionam muito bem. Outros não. Eles não são comuns nem populares na cultura ocidental.

Porém, às vezes, você é forçado a ir para a cama com uma linguagem de que você acha que não vai gostar somente para descobrir que, com tempo e experiência, isso é profundamente agradável.

Até que ponto a sua apreciação por uma linguagem molda a qualidade do código que você escreve ou a maneira como suas habilidades melhoram ao trabalhar com ela? Que parte disso nasce da aceitação, do respeito e do aumento de familiaridade? Entenda que amor e respeito podem se desenvolver com o tempo.

Compromisso

Um bom casamento exige *compromisso*: uma determinação para permanecer nele nos momentos bons e ruins em vez de pular fora quando a situação se tornar desconfortável.

Para se transformar em um programador expert em qualquer linguagem ou tecnologia, você deve se comprometer em aprendê-la, investir tempo e trabalhar com ela. Você não pode ser egoísta e esperar que *ela* satisfaça a todas as suas necessidades, especialmente quando ela tiver sido especificamente concebida para se adequar a muitas situações e a alguns requisitos diversificados.

Esse compromisso também pode significar que sacrifícios serão necessários. Você deverá abrir mão de algumas de suas maneiras preferidas de trabalhar para acomodar a outra parte. A linguagem tem idioms em particular e maneiras de trabalhar que são mais adequadas. Pode ser que você não goste deles ou que prefira trabalhar de outras maneiras. Contudo, se esses idioms forem a definição de um "bom" código, você deverá adotá-los.

O seu compromisso em escrever um bom código na linguagem atual supera o seu desejo de fazer algo de sua própria maneira? Um *bom código* ou uma *vida fácil*? Tudo tem a ver com compromisso.

> **PONTO-CHAVE** Para escrever o melhor código em uma linguagem, você deve se comprometer com seus estilos e os idioms em vez de se impor sobre ela.

Comunicação

Em um bom casamento, você deve se comunicar constantemente. Deve compartilhar fatos, sentimentos, mágoas e alegrias. Não fale apenas superficialmente, como faria com conhecidos com quem você se encontra na rua, mas no verdadeiro nível de coração para coração. É profundo. Você deve compartilhar fatos com o seu parceiro que não compartilharia com mais ninguém. Esse tipo de comunicação exige um nível muito elevado de confiança, de aceitação e de compreensão.

Isso não é necessariamente fácil; as pessoas se comunicam de maneiras

bem diferentes. A comunicação pode ser facilmente confundida ou mal interpretada. Para se comunicar com sucesso em um casamento é necessário um esforço enorme. É algo em que você deve prestar atenção e investir um esforço constante. A comunicação é mais uma habilidade que se adquire, e não apenas algo que você *pode* ou *não pode* fazer.

O ato da programação tem tudo a ver com comunicação. O código que escrevemos é tanto uma comunicação do propósito de nosso programa (para nós mesmos e para outros programadores que poderão usá-lo) quanto uma lista de instruções para um computador executar.

Nesse sentido, nós nos comunicamos tanto *com* a linguagem – para lhe dizer o que ela deve fazer de maneira clara, concisa, não ambígua e correta – quanto com *outras pessoas* que usam a linguagem de programação como meio.

Uma boa comunicação é uma habilidade vital (com frequência, ausente) em desenvolvedores de software de alta qualidade. Ela exige uma enorme quantidade de esforço e atenção constante para ser bem feita. Lembre-se de que a comunicação diz respeito tanto a ouvir quanto a falar.

> **PONTO-CHAVE** Bons programadores se comunicam bem. Eles falam, escrevem, codificam, ouvem e leem bem.

Paciência

Bons casamentos não surgem da noite para o dia. Eles são cultivados. E crescem. Gradualmente.

Em nossa cultura de *fast food*, aprendemos a esperar tudo para *já*: comida instantânea, dinheiro instantâneo, downloads instantâneos, gratificação instantânea. Porém os relacionamentos nunca funcionam dessa maneira.

O mesmo ocorre com o nosso relacionamento com a programação. Você pode saber da existência de uma linguagem em um instante. Pode até mesmo sentir uma atração instantânea: desejo de programação. Porém pode levar muito tempo para dominar totalmente uma linguagem, ser capaz de reivindicar honestamente que você realmente sabe como escrever um “bom” código com ela. Isso pode exigir bastante tempo e muita

paciência até que toda a beleza de uma linguagem possa ser apreciada.

> **PONTO-CHAVE** Não espere se tornar mestre de uma linguagem da noite para o dia e não fique frustrado enquanto estiver trabalhando nisso.

É claro que as linguagens mais agradáveis têm uma curva inicial suave de aprendizado, portanto você terá a impressão de estar fazendo bons progressos quando iniciar o seu relacionamento.

Valores compartilhados

Uma cola poderosa que mantém os relacionamentos unidos é ter uma noção comum de moral, de valores e de crenças. Por exemplo, pesquisas mostram que casais que compartilham fortes crenças religiosas tendem muito mais a permanecer unidos do que aqueles que não o fazem; elas atuam como uma fundação sólida sobre a qual o relacionamento é criado.

Se você não concordar com os valores básicos de uma linguagem – os muitos recursos e idioms que ela oferece –, seu relacionamento com a linguagem sempre será tortuoso.

Uma metáfora perfeita?

Isso é esclarecedor. Entretanto nenhuma metáfora é perfeita. A fidelidade à sua linguagem é tão importante para uma codificação saudável quanto o é para um casamento saudável? Não. Na verdade, é muito útil “pular a cerca” e se envolver com outras linguagens. Faça do C# a sua musa e use um pouco de Python lateralmente. Isso o fará adquirir algumas habilidades, aprender técnicas diferentes de programação e o ajudará a evitar que você fique preso a uma única trilha de programação.

Ou isso *é* exatamente como em um casamento? Deixarei que você decida.

Conclusão

Essa metáfora colorida relacionada ao casamento nos mostra que o conhecimento de uma linguagem não é tudo o que existe na programação. Considere como você trabalha com suas ferramentas – o

tipo de relacionamento que você tem com elas.

Bons programadores pensam mais do que apenas em linhas de código ou em design de código isolado. Eles se importam com o modo como usam e interagem com suas ferramentas e procuram descobrir como tirar o máximo de proveito delas, assim como se importam com simples conhecimentos factuais sobre elas.

Bons programadores não esperam respostas rápidas para corrigir problemas, mas aprendem a viver e a apreciar os pontos fortes e os pontos fracos de suas ferramentas. Eles se comprometem a viver com elas e a investir tempo e esforço para conhecê-las. Eles as apreciam e as valorizam.

Perguntas

1. Quais são as arestas a serem aparadas em sua linguagem atual? Liste seus pontos fortes e os pontos fracos.
2. Com quais outras linguagens e ferramentas você trabalha? Que nível de comprometimento você tem mostrado para conhecê-las intimamente?
3. Dizem que os casais ficam mais parecidos com o passar do tempo. Você se ajustou de modo a ser moldado pela sua linguagem? Foi para melhor ou para pior?
4. Quais linguagens mostram mais rapidamente sinais de negligência se os programadores não se comprometerem com elas? (Você chega em casa por meio de muitos ponteiros indiretos e encontra o seu objeto dentro do cachorro. E por mais que tente, você não consegue perceber *o que* fez para ofender a linguagem.)

Veja também

- *Importar-se com o código* (Capítulo 1) – Você se importa com as linguagens porque ama aprender e se importa com o código.
- *Viva para amar o aprendizado* (Capítulo 24) – Essas técnicas o ajudarão a conhecer novas linguagens.
- *Desenvolvimento de software é...* (Capítulo 14) – Mais metáforas para o desenvolvimento de software. (Caso elas tenham se esgotado.)

TENTE ISTO... Determine como aprofundar o seu relacionamento com a(s) sua(s) linguagem(ns) preferida(s).

---

1 São as linguagens em que você fornece uma lista serial de instruções que detalham *como* o código deve funcionar. Por outro lado, nas *linguagens declarativas*, você descreve *o que* deve ser feito. A linguagem descobre como.

2 N.T.: Problema que se previa que os sistemas de informática enfrentariam na passagem do ano de 1999 para o ano 2000.

3 N.T.: É uma ambiguidade que surge quando duas classes B e C herdam de A, e a classe D herda tanto da classe B quanto da classe C. Se houver um método em A que tenha sido sobrescrito por B e/ou por C, mas não por D, qual versão do método D herdará, de B ou de C? (Fonte: *http://en.wikipedia.org/wiki/Multiple_inheritance*)

4 Quantas pessoas jogam fora seus primeiros relacionamentos na esperança de trocá-los por algo que exija menos manutenção somente para descobrir que o modelo substituto é *tão* volúvel quanto o primeiro, igualmente difícil de conviver e não tão satisfatório?

CAPÍTULO 30

Postura dos programadores

Uma boa postura reflete um estado de espírito apropriado.

– Morihei Ueshiba

À medida que a pressão do projeto de desenvolvimento de software aumenta, a cobrança sobre os programadores cresce e passamos do tradicional dia de trabalho de 15 horas para mais próximos de 26 horas absurdas. Em um clima como esse, torna-se cada vez mais importante garantir que você tenha um ambiente de trabalho bastante confortável e ergonômico.

Essa talvez seja uma questão tão importante para o programador do século XXI quanto um bom design de código ou qualquer outra prática relacionada ao desenvolvimento de software. Afinal de contas, você não pode ser um desenvolvedor Agile com costas ruins; ninguém quer empregar um programador com *rigidez* muscular. E você não poderá navegar por um diagrama de classes UML complexo se não enxergar bem.

Para melhorar a qualidade de vida passada diante de um computador e garantir o seu bem-estar físico, neste capítulo, daremos uma olhada em como otimizar o seu ambiente de trabalho.

Preste bastante atenção; se você não entender esse assunto direito, poderá acabar com contas médicas altíssimas. Você me agradecerá um dia.

Postura básica diante do computador

Inicialmente, vamos dar uma olhada no caso mais básico de uso cotidiano de um computador: sentar-se diante de um monitor (ou, como os mais antigos dos departamentos de recursos humanos o chamavam: um “VDU”[1]). Provavelmente, você faz isso durante horas e horas em um dia,

portanto é vital garantir que você se sente da maneira correta. Surpreendentemente, sentar-se é uma tarefa bem complicada. Exige trabalho árduo e determinação para dominar. A prática leva à perfeição. À medida que prosseguir nesta seção, lembre-se de fazer pausas regulares – saia para dar uma caminhada ou faça algo igualmente relaxante.

A maneira como você se senta diante de um computador tem implicações não somente em sua produtividade (uma postura ruim pode ter um efeito significativo em sua concentração e, desse modo, em sua produtividade), mas também em sua saúde. Uma postura inadequada pode resultar em dores no pescoço, nas costas, dores de cabeça, problemas digestivos, dificuldades respiratórias, vista cansada... e a lista continua. Um exemplo de uma boa postura sentada está sendo mostrada na figura 30.1.

Figura 30.1 – Boa postura.

Estas são as recomendações ergonômicas dos especialistas:

- Ajuste a posição de sua cadeira e de seu monitor para que seus olhos estejam no mesmo nível da parte superior de sua tela e seus joelhos estejam um pouco abaixo de seus quadris. Ajuste o monitor para que esteja a uma distância confortável em relação a você (por exemplo, aproximadamente de 45 a 60 centímetros).
- Seus cotovelos devem permanecer em um ângulo de aproximadamente 90 graus. Você não deve mover seus ombros de modo significativo ao digitar ou usar o mouse. Para isso, o seu teclado deve estar aproximadamente na altura do cotovelo.
- O ideal é que o ângulo de seus quadris seja de 90 graus ou um pouco mais. (Você está pensando nisso enquanto lê, não está?)

- Seus pés devem permanecer retos no chão; não os coloque embaixo da cadeira. Tampouco sente-se sobre eles – você terá dores terríveis nas pernas e uma marca de sapato no traseiro.
- Seus pulsos devem descansar sobre a escrivaninha à sua frente. (Certamente, deixá-los em uma escrivaninha atrás de você seria uma péssima postura. A menos que você seja extremamente flexível.) Seus pulsos devem permanecer retos ao digitar.
- Ajuste sua cadeira para dar apoio à parte inferior das costas.

Para evitar problemas:

- Mude sua posição ao longo do dia para manter seus músculos relaxados e tirar a tensão de seu corpo.
- Faça várias pausas e caminhe pelo escritório. Você pode achar benéfico conversar com outras pessoas. Com um pouco de prática, a comunicação oral pode se tornar natural, até mesmo para um programador veterano.
- Não deixe o pescoço cair enquanto lê a tela. Mantenha a sua cabeça firme e tenha orgulho de ser um programador.
- Desfoque sua visão ocasionalmente. Tente aqueles desenhos com imagens 3D que eram populares nos anos 90. Ou mova o foco da tela para um objeto distante (talvez você possa olhar esperançosamente para a porta ou dar uma espiada nos prazos finais se perdendo a distância).
- Em casos realmente extremos de fadiga muscular, pode ser que seja necessário tomar uma atitude drástica: saia do prédio (sim, é perfeitamente seguro fazer isso) e dê uma volta mais longa. Se a caminhada for relaxante demais, você encontrará assentos suficientes no parque nas proximidades para poder praticar o ato de se sentar por um tempo.

> **PONTO-CHAVE** Cuide de si mesmo. Mantenha uma boa postura em seu trabalho.

Após ter determinado uma boa postura para o caso básico de uso do computador, vamos agora dar uma olhada em algumas das posturas menos consideradas, porém necessárias ao programador moderno. Afinal

de contas, é importante garantir que permaneceremos ergonomicamente adequados o dia *todo*.

A postura de debugging

O código acertou você de jeito? Os espíritos do mal se recusam a sair do lugar? Você está concentrado há seis horas seguidas e, apesar disso, ainda não conseguiu descobrir por que há um retângulo marrom horrível na tela, quando deveria haver um octógono turquesa elegante?

Nesse caso, o seu corpo precisa de uma postura levemente diferente para acomodar o peso do mundo em seus ombros e o deslocamento de seu córtex cerebral da parte superior de seu corpo para algum lugar dentro de seu sapato. Para suportar adequadamente o seu corpo e evitar mais cansaço (infelizmente, o cansaço mental é inevitável), siga estes passos:

- Incline-se levemente para a frente (um ângulo de 45 a 60 graus nos quadris é melhor).
- Coloque seus cotovelos sobre a escrivaninha diante de você (o ideal é que eles fiquem na posição de seus pulsos ao digitar).
- Estenda seus antebraços verticalmente para cima.
- Apoie a cabeça em seus mãos.
- Suspire.

A figura 30.2 ilustra essa postura. Nessas situações, talvez você ache mais confortável mover o monitor um pouco mais à frente da escrivaninha em relação à posição em que ele normalmente estaria. Você perceberá que isso facilitará bastante bater a sua cabeça contra ele repetidamente quando você se sentir particularmente frustrado.

Figura 30.2 – A postura da reflexão.

Quando a situação estiver realmente ruim

Apesar de adotar uma postura cuidadosa de debugging, talvez você não seja capaz de solucionar aquele problema espinhoso. Os bugs simplesmente não irão sumir. Por mais despreocupado que você queira parecer, eles simplesmente parecem não respeitar a sua postura determinada (embora confortável). A programação nem sempre é um mar de águas tranquilas e, às vezes, toda a baboseira de costas retas e ombros relaxados pode ir por água abaixo.

Se você descobrir que a situação *está* realmente ruim, adote a posição mostrada na figura 30.3 e prepare-se para o caso de tudo cair e explodir ao seu redor.

Figura 30.3 – A postura da aflição.

Para aqueles que trabalham a noite toda

Quando os prazos finais se assomarem, você poderá se ver trabalhando heroicamente durante horas para ter tudo concluído no prazo. É claro que você sabe que ninguém irá agradecê-lo por isso, porém um senso de obrigação moral e o orgulho por seu trabalho irão impeli-lo a permanecer acordado por três noites seguidas e viver com uma dieta à base de cafeína e de rosquinhas ressecadas.

Nessas situações, você verá que a postura da figura 30.4 é particularmente útil, em especial depois da quarta noite de trabalho. Como ocorre com qualquer outra consideração ergonômica, o aspecto realmente importante, nesse caso, é ajustar o seu ambiente de trabalho para ajudar você. Se possível, feche as persianas e a porta para bloquear qualquer barulho externo ou qualquer outro evento que possa distraí-lo de sua tarefa atual. Se você trabalha em uma área comunitária barulhenta, com várias pessoas transitando o dia todo, organize a sua escrivaninha e a cadeira com o máximo de potencial para não serem vistas.

Procure não roncar muito alto. Talvez seja útil inserir o mouse firmemente em sua boca para obstruir a passagem de ar. (Lembre-se de não fazer isso caso seu nariz esteja entupido, ou você poderá se asfixiar.)

Figura 30.4 – A postura do cochilo eficaz.

Intervenção do alto

Ocasionalmente, um chefe se sente impelido a rondar por aí para garantir que seus súditos estejam trabalhando tanto quanto burros de carga. Para

garantir o máximo conforto *dele* e evitar que ele distenda seus músculos delicados, você deve adotar a postura mostrada na figura 30.5. É para o bem dele:

- Empregue uma postura pesarosa, ensaiada. Retese todos os seus músculos e faça com que pareça que você está pronto para perseguir um assaltante.
- Adote uma expressão facial perturbada (se essa já não for a sua aparência natural após anos de programação). Algo na linha de uma constipação severa proporcionará uma aparência adequada de concentração extrema.
- Para ter um efeito melhor, compre um pouco de gelo seco (isso pode ser prontamente obtido em uma loja de produtos para shows) e deixe embaixo de sua escrivaninha. O chefe ficará impressionado com o aquecimento gerado pelo ritmo efervescente de seu trabalho. Contudo não fique tentado a exagerar, pois seus colegas poderão começar a se preocupar com seus problemas de flatulência ou a segurança poderá acionar o corpo de bombeiros.

O ideal é que o seu espaço de trabalho esteja orientado de modo que suas costas estejam contra uma parede; desse modo, ninguém poderá chegar pelas suas costas enquanto você estiver desprevenido. Adotar a postura mostrada na figura 30.5 rapidamente pode resultar em distensão muscular (especialmente se você precisar tirar o seu pé da escrivaninha às pressas) e lesões nos nervos.

Figura 30.5 – A postura "perfeita".

Voltando ao normal

Tome cuidado ao adotar a postura anterior para não virar demais os olhos. É importante poder ver o seu chefe se afastar para saber quando é seguro relaxar e adotar a postura da figura 30.6.

Você poderá descobrir que usar um joystick para jogos em rede exige menos esforço do pulso do que um teclado, portanto é preferível ter o primeiro se estiver disponível. Preencher um formulário de reembolso de maneira criativa deve permitir justificar a compra de um dispositivo para jogos de muito boa qualidade. Não considere controladores de Nintendo Wii no escritório – eles não são exatamente discretos.

Figure 30.6 – A postura da diversão.

É hora do design

Nossa última postura do programador deve ser empregada no design de um código novo ou quando estivermos trabalhando em problemas realmente difíceis. Nesses momentos, é importante garantir o máximo de conforto para não ser distraído pelo ambiente ao redor.

Você verá que a figura 30.7 é bastante autodescritiva.

Figura 30.7 – A postura do banheiro.

Vista cansada

Por fim, é importante reservar um pouco de tempo para considerar a saúde de seus olhos. Certifique-se de que, quando olhar para o monitor, você não estragará a sua visão. Faça pausas frequentes. Garanta que a sua tela não tenha reflexos excessivos de janelas ou de lâmpadas; mova a tela de lugar se isso for um problema. Certifique-se também de que não haja fontes diretas de luz (uma janela ou uma lâmpada) apontando diretamente para você.

De vez em quando, olhe melancolicamente pela janela para contemplar a alegria do mundo real lá fora.

Exames de vista regulares são essenciais. Aqui está um teste simples que você pode experimentar do conforto de sua cadeira giratória, que pode servir também como um bom exercício regular para os olhos. Imprima uma cópia da figura 30.8 e pendure-a na parede acima de sua escrivaninha (pode ser que seja necessário testar para saber qual é a melhor distância entre você e o cartaz). De vez em quando, durante o dia, transfira o seu foco da tela do computador para o cartaz. Comece lendo a letra na parte

de cima e continue compassadamente para baixo. Leia até o máximo que puder em direção à parte inferior.

Figura 30.8 – Teste ocular.

Conclusão

Sim, estou sendo irreverente. Entretanto esse *é* um assunto importante, sobre o qual você deve pensar. Muitos programadores não cuidam o suficiente de si mesmos fisicamente.

É vital garantir que sua estação de trabalho seja ergonomicamente adequada e que você não estragará a sua visão, não desenvolverá LER nem terá uma lesão nas costas por passar longos dias sentado, com os olhos fixos na tela de um computador. Você só tem um corpo: cuide dele.

Não, não sou seu pai reclamando de seu desleixo. (Mas tire o seu pé da escrivaninha – você está fazendo o local parecer desorganizado.)

Considere o uso de uma escrivaninha para trabalhar em pé (elas estão ficando bem populares atualmente). Certifique-se de que sua cadeira não seja uma porcaria qualquer, barata e horrível, mas um móvel adequadamente ajustável, com suporte apropriado para a lombar. Talvez você vá se beneficiar com um teclado e um mouse ergonômicos.

Faça pausas regulares. Permaneça hidratado durante o dia de trabalho. Evite a vista cansada com exercícios adequados aos olhos. Trabalhe uma quantidade razoável de horas e descanse o suficiente à noite.

Cuide de si mesmo!

Perguntas

1. O seu ambiente de trabalho tem boas instalações? É confortável? Você sente tensão enquanto trabalha?
2. Como o seu ambiente de trabalho pode ser melhorado? Por exemplo, seu monitor está a uma altura confortável? Sua cadeira é ajustável para que você possa manter os pulsos retos enquanto digita?
3. Quantas horas você trabalha por dia? Você trabalha mais horas para que o trabalho seja concluído? Que efeitos isso tem em seu corpo?
4. Você mantém um bom nível de hidratação enquanto trabalha? (Não beber líquido o suficiente leva a uma redução de sua capacidade de concentração.)

> **TENTE ISTO...** Avalie como a sua estação de trabalho está instalada. Tome medidas adequadas para evitar uma postura ruim e reduzir o cansaço ocular. Você só tem um corpo: cuide dele.

1 VDU: Vision Destruction Unit (Unidade de destruição da visão).

PARTE IV

Conseguir que tudo seja feito

A vida na produção de software pode ser agitada e ter um ritmo frenético, com muitas demandas absurdas. “Deixe isso extremamente elegante.” “Coloque mais recursos.” “Remova todos os bugs.” “E faça isso *agora*!” Com as pressões para prazos finais não realistas e tarefas complicadas de codificação se avolumando em sua cabeça, pode ser fácil perder o foco e entregar um software incorreto ou deixar de fazer a entrega.

Nos próximos capítulos, iremos explorar maneiras de criar códigos excelentes da melhor maneira possível – é a arte de fazer com que tudo seja feito.

CAPÍTULO 31

Mais inteligente, e não mais árduo

As batalhas são vencidas com assassinatos e manobras. Quanto melhor for o general, mais ele contribuirá com manobras e menos ele exigirá assassinatos.

– Winston Churchill

Deixe-me contar uma história a você. Ela é totalmente verdadeira. Um colega que estava trabalhando com um código de UI precisava colocar setas arredondadas bonitas em sua tela de exibição. Depois de ter brigado para fazer isso por meio de programação usando as primitivas de desenho disponíveis, sugeri que ele simplesmente sobrepusesse uma imagem na tela. Seria uma implementação muito mais simples.

E assim ele partiu para o trabalho. Abriu o Photoshop. E trabalhou. E ajustou. E trabalhou um pouco mais. Nesse aplicativo – o Rolls-Royce das aplicações para composição de imagens – não há nenhuma maneira rápida e fácil de desenhar uma seta arredondada que seja apenas meio decente. Presumivelmente, um artista gráfico experiente poderia criar uma em dois minutos. Porém, depois de quase uma hora desenhando, cortando, fazendo composições e reorganizando, ele ainda não tinha uma seta arredondada convincente.

Frustrado, ele mencionou esse fato a mim enquanto foi pegar uma xícara de chá.

Ao retornar, com o chá na mão, ele encontrou uma imagem nova e reluzente de uma seta arredondada em seu desktop, pronta para uso.

"Como você fez isso tão rapidamente?", ele perguntou.

"Só usei a ferramenta certa", respondi, esquivando-me de uma xícara de chá voadora.

O Photoshop *deveria* ter sido a ferramenta certa. É aquela com a qual a

maioria dos trabalhos de criação de imagens é feita. Mas eu sabia que o Open Office disponibilizava uma ferramenta prática e configurável para setas arredondadas. Eu a desenhei em dez segundos e enviei-lhe uma imagem de tela. Não era elegante. Mas funcionava.

Moral da história?

Há um perigo constante em focar demais em uma ferramenta ou em uma única abordagem para solucionar um problema. É tentadoramente fácil perder horas de esforço explorando seus becos escuros quando há um caminho mais fácil e mais direto até o seu objetivo.

Então como podemos melhorar isso?

Escolha suas batalhas

Para ser um programador produtivo, você deve aprender a trabalhar de forma *mais inteligente*, e não do modo *mais árduo*. Uma das marcas registradas dos programadores experientes não é somente a sua perspicácia técnica, mas o modo como eles resolvem problemas e escolhem suas batalhas.

Bons programadores fazem as tarefas rapidamente. Porém eles *não* arruínam o código como se fossem um cowboy atirando sem tirar a arma da cintura. Eles simplesmente trabalham de modo mais inteligente. Isso não ocorre necessariamente porque são mais inteligentes; eles simplesmente sabem como resolver *bem* os problemas. Eles têm um arsenal de experiência em que se basear, que os orientarão em direção à abordagem correta. Podem ver soluções alternativas – a aplicação de uma técnica incomum que fará com que o serviço seja feito com menos dores de cabeça. Sabem como traçar um caminho ao redor dos obstáculos que surgirem. Podem tomar decisões bem fundamentadas sobre os pontos em que será melhor investir esforços.

Táticas de batalha

A seguir, temos algumas ideias simples que ajudarão você a trabalhar de modo mais inteligente.

Reutilize sabiamente

Não escreva uma massa de código por conta própria se puder usar uma biblioteca existente ou puder adaptar um código de outro local.

Mesmo que seja preciso pagar por uma biblioteca de terceiros, normalmente, em termos de custo, é muito mais eficiente usar uma implementação pronta do que escrever o seu próprio código. E fazer os seus próprios testes. E então efetuar o seu próprio debug.

> **PONTO-CHAVE** Utilize um código existente em vez de fazer a sua própria implementação do zero. Empregue o seu tempo em tarefas mais importantes.

Supere a síndrome do "*não inventado aqui*". Muitas pessoas acham que podem fazer um trabalho muito melhor por conta própria ou criar uma versão mais apropriada para a sua aplicação específica. Esse é *realmente* o caso? Mesmo que o outro código não esteja projetado exatamente de acordo com a sua preferência, simplesmente use-o. Você não precisará necessariamente reescrevê-lo se ele já estiver funcionando. Crie uma fachada em torno dele caso precise integrá-lo ao seu sistema.

Faça com que o problema seja de outra pessoa

Não tente descobrir como realizar uma tarefa por conta própria se outra pessoa já souber como fazê-lo. Talvez você aprecie a glória da realização. Pode ser que você goste de aprender algo novo. Porém, se outra pessoa puder fazer com que você saia em vantagem na frente ou puder realizar a tarefa muito mais rapidamente, pode ser que seja melhor colocar a tarefa em sua fila.

Faça somente o que for necessário

Considere este sacrilégio: você *precisa* refatorar? Você *precisa* realizar teste de unidade?

Sou um firme defensor de ambas as práticas, mas, às vezes, elas podem não ser apropriadas ou podem não valer o investimento de seu tempo. Sim, sim: tanto refatorar quanto realizar testes de unidade trazem grandes benefícios e jamais devem ser deixados de lado descuidadamente.

Entretanto, se você estiver trabalhando em um protótipo pequeno ou estiver explorando um possível design funcional usando um código descartável, pode ser que seja melhor deixar as práticas teológicas corretas para mais tarde.

Se você investir tempo (louvavelmente) em testes de unidade, considere exatamente *quais* testes devem ser escritos. Uma abordagem inflexível do tipo "testar todos os métodos" não será sensata. (Normalmente, você achará que tem uma melhor cobertura do que o esperado). Por exemplo, você deve testar todo e qualquer getter e setter de sua API.[1] Foque seus testes no uso, e não nos métodos, e preste atenção em particular nos lugares em que você espera que vá haver mais problemas.

Escolha suas batalhas de testes.

Use uma solução rápida

Se você tiver várias opções de design e não estiver certo da solução a ser escolhida, não perca horas cogitando qual delas é a melhor. Uma solução *rápida* (um protótipo descartável) poderá gerar respostas mais úteis em minutos.

Para que isso funcione bem, defina um intervalo de tempo específico do tipo Pomodoro (*http://pomodorotechnique.com/*), em que a solução rápida será desenvolvida. Pare quando o tempo se esgotar. (E no verdadeiro estilo Pomodoro, adquira um bom *timer* difícil de ser ignorado para forçar você a parar.)

Utilize ferramentas que o ajudarão a voltar atrás rapidamente (por exemplo, um sistema de controle de versões eficiente).

Priorize

Priorize a sua lista de tarefas. Faça as tarefas mais importantes antes.

> **PONTO-CHAVE** Concentre seus esforços nas tarefas mais importantes antes. O que é mais urgente ou irá gerar mais valor?

Seja rigoroso em relação a isso. Não se deixe levar por minúcias que não sejam importantes; é extremamente fácil fazer isso. Em especial, quando

uma tarefa simples acaba dependendo de outra tarefa simples. Que, por sua vez, depende de outra tarefa simples, que depende de ... Depois de duas horas, você sairá de uma toca de coelho e estará se perguntando por que cargas-d'água você está reconfigurando o servidor de emails em seu computador quando o que você queria fazer era modificar um método de uma classe contêiner. No folclore computacional, isso se chama *yak shaving*[2].

Tome cuidado com as várias tarefas pequenas que você faz e que não sejam tão importantes: email, papelada, telefonemas – a burocracia. Em vez de fazer tudo isso ao longo do dia, interrompendo e distraindo você do fluxo de tarefas importantes, reúna-as e faça-as em um horário predeterminado (ou em alguns) todos os dias.

Pode ser que você ache útil anotar essas tarefas em uma pequena lista de "tarefas pendentes" e, em um determinado horário, comece a processá-las o mais rapidamente possível. Riscá-las de sua lista pode proporcionar um senso de realização bastante motivador.

O que é realmente necessário?

Quando uma nova tarefa for atribuída a você, verifique se ela é *realmente* necessária agora. O que o cliente precisa que você entregue?

Não implemente a versão Rolls-Royce, cheia de acessórios, se não for necessário. Mesmo que a tarefa solicitada exija, pare e verifique o que é realmente necessário. Para isso, você deve conhecer o contexto em que o seu código se encontra.

Não é apenas preguiça. Há um perigo em escrever código demais muito cedo. *O princípio de Pareto*[3] implica que 80% dos benefícios exigidos poderiam ser provenientes de apenas 20% da implementação pretendida. Você realmente precisa escrever o restante desse código, ou o seu tempo poderia ser mais bem empregado em outra atividade?

Uma tarefa de cada vez

Faça uma tarefa de cada vez. É difícil focar em mais de uma tarefa de uma só vez (especialmente para os homens, com seus cérebros para uma só

tarefa). Se tentar trabalhar de modo concorrente, ambas as tarefas serão cumpridas inadequadamente. Termine uma tarefa e então passe para a próxima. Você terá as duas tarefas concluídas em um período de tempo menor.

Mantenha o seu código pequeno (e simples)

Mantenha o seu código e o design o menor e o mais simples possíveis. Caso contrário, você irá simplesmente adicionar muito mais código, que lhe custará tempo e esforço para ser mantido no futuro.

> **PONTO-CHAVE** Lembre-se do princípio KISS: Keep It Simple, Stupid (Mantenha a simplicidade, seu estúpido).

Você *deverá* mudar o código; não será possível prever exatamente quais são os requisitos futuros. Prever o futuro é uma ciência incrivelmente inexata. É mais fácil e mais inteligente deixar o seu código maleável para mudanças agora do que incluir suporte para todas as possíveis funcionalidades futuras no primeiro dia.

Um corpo de código pequeno e focado é muito mais fácil de ser alterado do que um corpo grande.

Não adie os problemas deixando que eles se acumulem

Algumas tarefas difíceis (como integração de código) *não* devem ser evitadas porque são difíceis. Muitas pessoas fazem isso; elas adiam essas tarefas para tentar minimizar o sofrimento. Isso soa como se você estivesse escolhendo suas batalhas, certo?

Na verdade, a opção mais inteligente é começar o mais cedo possível e enfrentar a dor enquanto ela for menos intensa. É mais fácil integrar porções pequenas de código mais cedo e, então, integrar as alterações subsequentes com frequência do que trabalhar em três funcionalidades grandes durante um ano e tentar reuni-las no final.

O mesmo vale para os testes de unidade: escreva os testes agora, juntamente com o seu código (ou antes). Será muito mais difícil e menos produtivo esperar até que o código esteja “funcionando” para escrever os

testes.

Como diz o ditado, *se for difícil, faça com mais frequência*.

Automatize

Lembre-se do conselho clássico: *se tiver de fazer mais de uma vez, crie um script que faça por você.*

> **PONTO-CHAVE** Se você executar algo com frequência, faça o computador fazê-lo por você. Automatize usando um script.

Automatizar uma tarefa comum e enfadonha pode fazer você economizar muitas horas de esforço. Considere também uma única tarefa que tenha um alto grau de repetição. Pode ser que seja mais rápido criar uma ferramenta e executá-la uma vez do que realizar a tarefa repetitiva manualmente por conta própria.

Essa automação tem uma vantagem adicional: ela ajuda outras pessoas a trabalharem de forma mais inteligente também. Se você puder gerar o seu build com *um* comando no lugar de dar uma série de 15 comandos complexos e efetuar vários cliques de botão, sua equipe toda poderá gerar builds mais facilmente e os novatos poderão entrar em ação rapidamente.

Para ajudar nessa automação, os programadores experientes naturalmente escolherão ferramentas que possam ser automatizadas, mesmo que eles não pretendam automatizar nada nesse momento. Favoreça fluxos de trabalho que gerem textos simples ou arquivos intermediários estruturados de modo simples (por exemplo, JSON ou XML). Selecione ferramentas que tenham uma interface de linha de comando, bem como (ou no lugar de) um painel GUI inflexível.

Pode ser difícil saber se vale a pena criar um script para uma tarefa. Obviamente, se é provável que você vá realizar uma tarefa diversas vezes, então vale a pena considerar. A menos que o script seja particularmente difícil de escrever, é pouco provável que você desperdice tempo ao escrevê-lo.

Evitando erros

Encontre erros mais cedo para não passar muito tempo fazendo algo errado.

Para isso:

- Mostre o seu produto aos clientes bem cedo e com frequência para descobrir rapidamente se você não está construindo o sistema errado para eles.
- Discuta o design de seu código com outras pessoas para descobrir mais cedo se há uma maneira melhor de estruturar a sua solução. Não invista esforços em código ruim se puder evitar.
- Faça revisões de código em porções pequenas e compreensíveis de trabalho, e não em partes grandes e densas.
- Faça testes de unidade no código desde o início. Garanta que os testes de unidade sejam executados frequentemente para identificar erros antes que esses o prejudiquem.

Comunique-se

Aprenda a se comunicar melhor. Aprenda a fazer as perguntas certas para entender de forma não ambígua. Um erro de compreensão agora pode significar que você acabará fazendo retrabalhos em seu código futuramente. Ou que sofrerá atrasos à espera de mais respostas para perguntas essenciais. Isso é particularmente importante, e temos um capítulo todo sobre comunicação (Capítulo 36).

Aprenda a conduzir reuniões produtivas para que sua vida não seja sugada pelos demônios que se sentam nos cantos das salas de reuniões.

Evite um esgotamento

Não se deixe esgotar trabalhando em horários malucos, fazendo com que as pessoas esperem níveis não realistas de trabalho de você o tempo todo. Deixe claro se você estiver indo além da obrigação para que as pessoas aprendam a não esperar isso com frequência demais.

Projetos saudáveis não exigem excessos de horas extras.

Ferramentas poderosas

Sempre procure novas ferramentas que deem um impulso ao seu fluxo de trabalho.

Porém não se torne um escravo à procura de novos softwares. Geralmente, softwares novos têm arestas que podem feri-lo. Favoreça as ferramentas testadas e comprovadas, que várias pessoas já tenham usado. É impossível estabelecer um preço pelo conhecimento reunido sobre essas ferramentas, disponíveis por meio do Google.

Conclusão

Escolha suas batalhas (Sim, sim.) *Trabalhe de forma mais inteligente, e não do modo mais árduo!* (Já ouvimos tudo isso antes.)

São máximas triviais. Porém são verdadeiras.

É claro que isso não significa *deixar de trabalhar arduamente*. A menos que você queira ser demitido. Mas isso não é ser inteligente.

Perguntas

1. Como você pode determinar a quantidade certa de testes a ser aplicada em seu trabalho? Você conta com experiência ou com diretrizes? Relembre o trabalho de seu último mês; ele foi testado adequadamente?
2. Até que ponto você é bom em priorizar a sua carga de trabalho? Como você pode melhorar?
3. Como você garante que encontrará os problemas o mais rápido possível? Quantos erros você precisou corrigir ou quantos retrabalhos foram necessários que poderiam ter sido evitados?
4. Você sofre da síndrome do *não inventado aqui*? O código de todas as outras pessoas é ruim? Você poderia fazer melhor? Você consegue suportar o fato de o trabalho de outras pessoas ser incorporado ao seu?
5. Se você trabalha em uma cultura que valorize a quantidade de horas trabalhadas em detrimento da qualidade desse trabalho, como o fato

de trabalhar pode ser conciliado com “trabalhar de forma inteligente” sem que você pareça preguiçoso?

Veja também

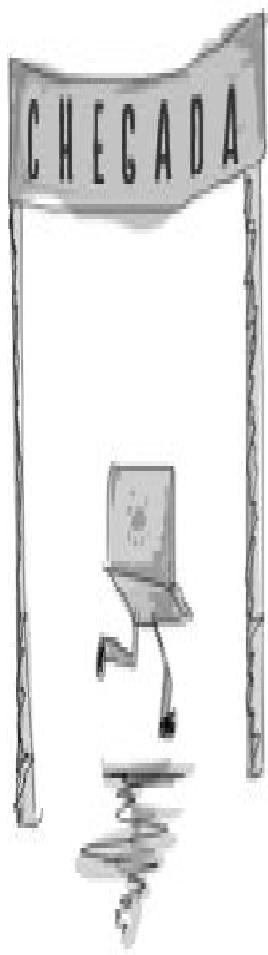
- *Dessa vez, eu consigo* (Capítulo 33) – Uma história sobre precaução: é fácil deixar de trabalhar de forma tão inteligente quanto você é capaz.
- *Um estudo sobre reutilização de código* (Capítulo 19) – Empregue a abordagem “inteligente” na reutilização de código. Não faça uma confusão com duplicações, e não crie mais código do que o necessário.
- *Estará pronto quando estiver pronto* (Capítulo 32) – Não faça mais trabalhos além do necessário – aprenda a definir quando uma tarefa está “pronta”.

> **TENTE ISTO...** Identifique três técnicas que ajudarão você a se tornar um programador *mais produtivo*. Tenha como meta adotar duas novas práticas e parar de usar outra. Comece a empregar essas técnicas amanhã. Torne-se responsável em relação a elas diante de alguém.

---

1 O fato de que você deva ter getters e setters em suas APIs, antes de tudo, é outro problema.

2 N.T.: Barbear um iaque, em uma tradução literal. A expressão quer dizer realizar atividades sem sentido.

3 Para muitos eventos, de modo geral, 80% dos efeitos são provenientes de 20% das causas. Para mais informações sobre esse assunto, acesse *http://en.wikipedia.org/wiki/Pareto_principle*.

CAPÍTULO 32

Estará pronto quando estiver pronto

Em nome de Deus, pare um instante, interrompa o seu trabalho, olhe ao seu redor.

– Leon Tolstói

Um programa é constituído de vários subsistemas. Cada um desses subsistemas é composto de partes menores – componentes, módulos, classes, funções, tipos de dados e partes afins. Às vezes, até mesmo de caixas e de linhas. Ou de ideias inteligentes.

O programador que trabalha se move de uma incumbência para a

próxima; de uma tarefa para outra. O dia de trabalho é composto de uma série de tarefas de construção e de manutenção em uma série de componentes desse software: compondo partes novas, unindo partes, estendendo, melhorando ou corrigindo partes existentes do código.

Desse modo, o nosso trabalho é simplesmente uma sequência de vários trabalhos menores. É recursivo. Os programadores amam esse tipo de ideia.

Já terminamos?

Então aí está você, fazendo com que o trabalho fique pronto. (Ou é o que você pensa.)

Assim como uma criança pequena viajando no banco traseiro de um carro berrando constantemente *já chegamos?*, logo você irá se deparar com o gerente gritando: *já terminamos?*

Essa é uma pergunta importante. É essencial que um desenvolvedor de software seja capaz de responder a essa pergunta simples: saber o que quer dizer "pronto" e ter uma ideia realista do quanto você está próximo de "pronto". E então informar isso.

Muitos programadores deixam a desejar nesse caso; é tentador simplesmente ficar codificando até a tarefa parecer completa. Eles não sabem muito bem se estão próximos ou não de terminar. Eles pensam: *pode haver qualquer quantidade de bugs para corrigir ou problemas imprevistos em que posso tropeçar. Não tenho como dizer se está quase pronto.*

Porém isso simplesmente não é bom o suficiente. Em geral, evitar a pergunta é uma desculpa para uma prática preguiçosa; é uma justificativa para codificar de forma desleixada, sem pensar antes ou planejar. Não é metódico.

É provável que isso também crie problemas para você. Com frequência, vejo pessoas trabalhando de forma excessivamente árdua:

- Elas fazem mais trabalho do que o necessário porque não sabem quando devem parar.

- Sem saber quando terminarão, elas não *completam* realmente as tarefas que acham que estão terminadas. Isso resulta na necessidade de retomar o código posteriormente, descobrir o que está faltando e incluí-lo. A implementação do código é muito mais lenta e difícil dessa maneira.
- As partes incorretas do código são polidas, pois o objetivo correto não estava visível. É um trabalho desperdiçado.
- Os desenvolvedores que trabalham demais são forçados a fazer horas extras. Você não dormirá o suficiente!

Vamos ver como evitar isso para responder a "já chegamos?" de maneira eficiente.

Desenvolvendo para trás: decomposição

Diferentes empresas de programação administram seus esforços diários de desenvolvimento de modo diferente. Geralmente, isso depende do tamanho e da estrutura da equipe de software.

Algumas colocam um único desenvolvedor como responsável por um grande conjunto de funcionalidades, informam-lhe uma data de entrega e pedem relatórios ocasionais sobre o progresso. Outras seguem processos mais ágeis e administram um backlog de tarefas granulares (talvez chamando-as de *histórias*), dividindo-as entre os programadores à medida que eles puderem assumir uma nova tarefa.

O primeiro passo em direção à definição de "pronto" é saber *exatamente* em que você está trabalhando. Se for um problema diabolicamente grande e complexo, então será diabolicamente complexo dizer quando ele estará resolvido.

Responder em que ponto você está em um problema pequeno e bem compreendido é um exercício muito mais simples. Isso é óbvio.

Portanto, se você foi alocado para uma tarefa monstruosa, antes de começar a trabalhar nela, divida-a em partes menores e compreensíveis. Muitas pessoas se apressam e entram de cabeça no código ou no design sem parar para considerar como irão trabalhar na tarefa.

PONTO-CHAVE Divida tarefas grandes em uma série de tarefas menores e bem compreendidas. Você poderá julgar o progresso por meio delas de maneira mais precisa.

Geralmente, essa não é uma tarefa complexa, pelo menos para uma decomposição de alto nível. (Pode ser que você tenha de detalhar mais algumas vezes. Faça isso. Porém preste atenção: esse é um indício de que você recebeu uma tarefa com uma granularidade muito elevada.)

Às vezes, uma decomposição como essa é difícil de ser feita e, por si só, é uma tarefa significativa. Não deixe isso desanimá-lo. Se não fizer isso com antecedência para poder dar estimativas, você acabará fazendo posteriormente, de maneiras menos focadas, à medida que estiver batalhando para alcançar a linha de chegada.

Certifique-se de que, a qualquer momento, você saiba qual é a menor unidade em que você está trabalhando, em vez de saber somente qual é o grande alvo de seu projeto.

Defina "pronto"

Você tem uma ideia do quadro geral; sabe o que está tentando criar, em última instância. E sabe qual é a subtarefa em particular em que está trabalhando no momento.

Certifique-se de que, para qualquer tarefa em que estiver trabalhando, você saiba *quando deve parar*.

Para isso, você deve definir o que significa "pronto". Deve saber o que significa "sucesso" e qual será a aparência do software "completo".

PONTO-CHAVE Não se esqueça de definir o que é "pronto".

Isso é importante. Se você não determinar o momento em que deve parar, você acabará trabalhando muito mais do que o necessário. Você trabalhará mais e de forma mais árdua que o necessário. Ou não trabalhará de modo suficientemente árduo – nem tudo ficará pronto. (Não ter *tudo* pronto parece mais fácil, não é mesmo? Mas não é... o trabalho pela metade virá assombrá-lo de volta e representará mais

trabalho no futuro, seja na forma de bugs, de retrabalhos ou de um produto instável.)

Não comece um trabalho de codificação até saber o que é sucesso. Se ainda não souber, faça com que sua primeira tarefa seja determinar o que significa "pronto". Com muita frequência, não é o programador que define isso, mas o dono de um produto, o designer do sistema, o cliente ou o usuário final.

Somente então prossiga. Com a certeza de saber a direção para a qual você está seguindo, será possível trabalhar de maneira focada e direcionada. Você poderá fazer escolhas bem fundamentadas e deixar de lado itens desnecessários que poderão desviá-lo ou atrasá-lo.

> **PONTO-CHAVE** Se você não puder dizer quando uma tarefa estará pronta, não a comece.

Como isso é feito na prática? Como você define "pronto"? Os seus critérios de "pronto" devem ser:

Claros

Os critérios devem ser específicos e não devem ser ambíguos. Uma lista deve conter todas as funcionalidades a serem implementadas, as APIs a serem adicionadas ou estendidas ou as falhas específicas a serem corrigidas.

Se, à medida que se envolver com a tarefa, você descobrir fatos que possam afetar os critérios de conclusão (por exemplo, você descobre mais bugs que devam ser corrigidos, ou encontra problemas imprevistos), certifique-se de que isso se refletirá em seus critérios de "pronto".

Esses critérios normalmente podem ser diretamente relacionados a alguns requisitos de software ou a uma história de usuário – se você os tiver. Se for esse o caso, não se esqueça de documentar essa conexão.

Visíveis

Certifique-se de que os critérios de sucesso sejam vistos por todas as partes importantes. Provavelmente, elas incluem o seu gerente, seus clientes, as equipes que usarem o seu código depois de você ou os

responsáveis pelos testes, que validarão o seu trabalho.

Garanta que todos conheçam e concordem com esses critérios. E garanta que todos eles tenham uma maneira de dizer – e de concordar – em relação a quando sua tarefa está “pronta”.

Atingível

Defina cuidadosamente os critérios de “pronto”. Uma definição de “pronto” que seja inalcançável não tem utilidade: se estiver fora do alcance da equipe atual, será uma pedra em seu sapato em vez de ser um objetivo em direção ao qual ela se esforçará. Por exemplo, um objetivo de cobertura de 100% do código em um ambiente com poucos testes não é realista.

A natureza de cada tarefa claramente definirá o que “pronto” quer dizer. Entretanto você deve considerar:

- A quantidade de código que deve ser completada. (Você mede isso em unidades de funcionalidades, APIs implementadas ou histórias de usuário concluídas?)
- Quanto do design está pronto e como isso é identificado.
- Se algum documento ou relatório deve ser gerado.

Quando for uma tarefa de codificação, será possível demonstrar claramente que ela está “ficando pronta” por meio da criação de um conjunto de testes não ambíguo. Escreva testes que mostrem quando o conjunto todo de código necessário foi implementado.

> **PONTO-CHAVE** Use testes escritos no código para definir quando ele está completo e funcionando.

Há outras perguntas que você poderá ter de considerar ao descrever o que quer dizer “pronto”:

- Em que local o código será disponibilizado? (No sistema de controle de versões, por exemplo.)
- Em que local o código será implantado? (Ele estará “pronto” quando estiver ativo em um servidor? Ou você disponibiliza um produto testável, pronto para a equipe de implantação instalar?)

- Quais são os aspectos econômicos de “pronto”? Ou seja, os números exatos necessários que possam levar a determinados compromissos ou a certas medidas. Por exemplo, até que ponto a sua solução deve escalar? O seu software não será bom o suficiente se ele administrar somente dez usuários simultâneos se dez mil forem necessários. Quanto mais precisos forem os seus critérios de pronto, melhor você entenderá esses aspectos econômicos.
- Como você indicará que a sua tarefa está pronta? Quando você achar que está pronto, como fará para que o cliente/gerente/departamento de QA saiba? Provavelmente, isso será diferente para cada pessoa. Como você fará com que todos concordem que sua tarefa realmente está pronta – quem assina embaixo de seu trabalho? Você simplesmente faz check-in, altera um relatório de informações do projeto ou cria uma fatura?

Simplesmente faça

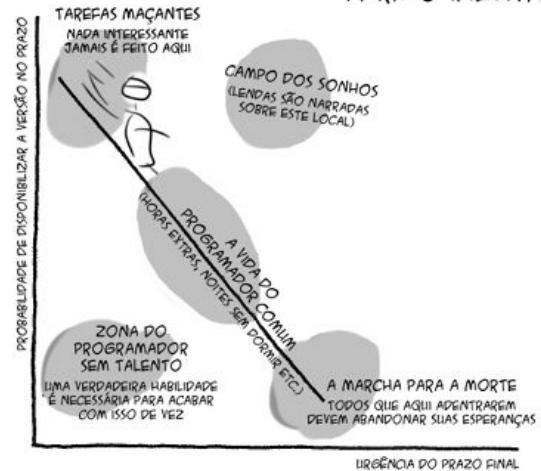
Ao definir o que é “pronto”, você poderá trabalhar com mais foco. Trabalhe em direção ao ponto que determine o que é “pronto”. Não faça mais do que o necessário.

Pare quando o seu código estiver bom o suficiente – e não *necessariamente* perfeito (pode haver uma verdadeira diferença entre os dois estados). Se o código for muito usado ou bastante trabalho tiver sido feito nele, talvez ele deva ser refatorado no futuro para estar perfeito – não faça nenhum polimento ainda. Isso pode ser simplesmente um desperdício de esforço. (Tome cuidado: isso não é uma desculpa para escrever um *código ruim*; é somente um aviso contra polir o código de maneira exagerada sem que haja necessidade.)

PONTO-CHAVE Não trabalhe mais do que o necessário. Trabalhe até estar “pronto”. Então pare.

Ter um único objetivo específico em mente ajuda a focar em uma única tarefa. Sem esse foco, é fácil trabalhar no código aleatoriamente, tentando atingir vários objetivos, sem administrar nenhum deles com sucesso.

Perguntas

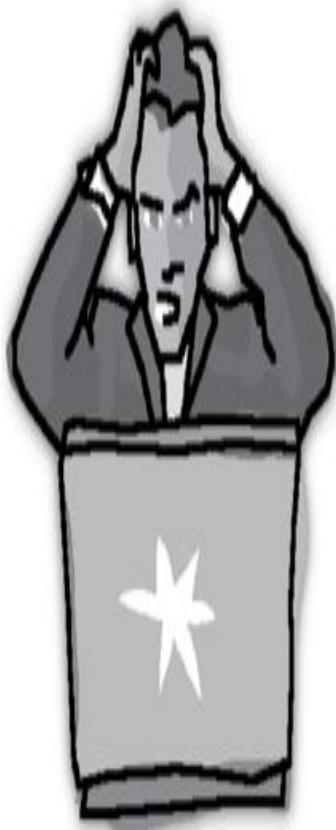
1. Você sabe quando a sua tarefa atual estará "pronta"? Qual é a aparência de "pronto"?
2. Você fez a decomposição de sua tarefa atual em um único objetivo ou em uma série de objetivos simples?
3. Você fez a decomposição de seu trabalho em unidades alcançáveis e mensuráveis?
4. Como o seu processo atual de desenvolvimento determina a forma de dividir e de estimar o trabalho? É suficiente?
5. Que nível de variação há quanto à precisão das estimativas feitas pelas pessoas de sua equipe? Por que você acha que isso acontece? O que faz com que as pessoas façam estimativas melhores?

Veja também

- *Mais inteligente, e não mais árduo* (Capítulo 31) – Defina "pronto" e não faça nada a mais. Isso é trabalhar de forma mais *inteligente*, e não mais *árdua*.
- *Nada está gravado a ferro e fogo* (Capítulo 18) – Nenhum software jamais estará totalmente "pronto". Por definição, o software é *soft*, os requisitos podem mudar amanhã e exigir que o alteremos.

> **TENTE ISTO...** Analise a sua tarefa atual de codificação. Ela tem o tamanho certo? Está decomposta corretamente? Defina um ponto claro que indique "pronto". Descubra como monitorar o seu progresso por meio dele de forma mais precisa.

10.000 MACACOS
(OU ALGO POR AÍ)
O PROGRAMADOR ESCORREGA PARA O INEVITÁVEL
TAREFAS MAÇANTES
NADA INTERESSANTE JAMAIS É FEITO AQUI
CAMPO DOS SONHOS
(LENDAS SÃO NARRADAS SOBRE ESTE LOCAL)
A VIDA DO PROGRAMADOR COMUM
(HORAS EXTRAS, NOITES SEM DORMIR ETC.)
ZONA DO PROGRAMADOR SEM TALENTO
UMA VERDADEIRA HABILIDADE É NECESSÁRIA PARA ACABAR COM ISSO DE VEZ
A MARCHA PARA A MORTE
TODOS QUE AQUI ADENTRAREM DEVEM ABANDONAR SUAS ESPERANÇAS
PROBABILIDADE DE DISPONIBILIZAR A VERSÃO NO PRAZO
URGÊNCIA DO PRAZO FINAL

10.000 MACACOS
(OU ALGO POR AÍ)

CAPÍTULO 33

Dessa vez, eu consigo...

É mais fácil evitar maus hábitos do que deixá-los de lado.

– Benjamin Franklin

“Só mais um minuto”, disse Jim. “Acho que agora sei realmente qual é o problema. Dessa vez, vou corrigi-lo.”

Julie observara Jim tentando resolver o problema durante quase o dia todo e estava cada vez mais impressionada.

Ele havia ficado sobre o seu teclado durante horas seguidas. Jim mal olhava para cima. Com certeza, ele não tinha comido. E havia tomado somente um café que Julie lhe trouxera no meio da manhã, mais por estar com pena dele.

Não parecia nem um pouco ser o Jim. Era um homem em uma missão.

Um senso de urgência, se não de leve pânico, havia se instalado por causa de um bug de "nível 1" descoberto no sistema ativo. Como isso havia passado pelo processo de QA era o que todos queriam saber.

Imaginava-se que era um problema em alguma parte do código de Jim e, sendo assim, ele havia entrado em ação. Em parte, foi o orgulho que o impediu de pedir ajuda, porém houve também uma ponta de ingenuidade – ele achou que iria identificar o problema em dez minutos e então seria um herói por corrigir o sistema em execução.

Até então, esse plano havia falhado.

A cada minuto que passava, a pressão aumentava. Estavam chovendo relatórios dos clientes a respeito do problema. Um ou dois relatórios logo de manhã haviam se transformado em um fluxo contínuo. Não muito tempo depois, esse fluxo se transformou em uma enchente e a equipe toda seria jogada nela. Com efeito, se o problema não fosse corrigido logo, a empresa iria sofrer as consequências.

Ninguém queria isso em sua consciência.

Ou em seus currículos.

Jim precisava corrigir isso. E rápido. A pressão estava aumentando.

A essa altura, eles certamente deviam ter restaurado o código para uma versão boa já conhecida e conseguido mais tempo para fazer o diagnóstico e a correção, porém, em alguns momentos, Jim garantia a Julie que ele estava "quase lá". E ele realmente acreditava nisso. Contudo, a cada vez que Jim se aproximava da causa do problema, quando ele achava que o havia cercado, parece que o problema se deslocava para um canto mais escuro do sistema.

Estava claro agora que o problema não estava somente no código de Jim. Todos os seus testes de unidade mostravam que o módulo funcionava tão

bem quanto esperado. Não, era um problema complicado de integração; algo estranho estava acontecendo na fronteira de vários módulos de software. E era intermitente também. O problema estava relacionado com questões sutis de tempo ou com a ordem dos eventos fluindo pelo sistema.

A presa de Jim, como um cervo tímido, estava sumindo de sua vista. Ele simplesmente não conseguia encontrá-la.

"Acho que agora eu sei onde está o problema. Não é no dispatcher de eventos em si. Acho que há algo estranho na comunicação entre ele, o banco de dados e o backend fazendo o processamento", disse Jim. "Cheguei a esses três componentes. Ainda não fiz a correção, mas a próxima alteração realmente *deverá* funcionar." Ele tentava parecer mais seguro do que realmente estava.

"É mesmo? Você tem certeza?", perguntava Julie. Havia certo ar de escárnio em sua voz. Isso não passou despercebido. Normalmente, Jim entraria na brincadeira, porém ele não estava disposto nesse dia. Ele lhe lançou um olhar normalmente reservado aos fiscais de trânsito e voltou a observar as várias janelas de código-fonte abertas em sua tela.

"Se eu apenas pudesse..."

"Espere um minuto", interrompeu Julie. "É sério, apenas *espere*. Pare e pense no que você está fazendo." Sua voz calma atingiu Jim e ele olhou para cima novamente. Jim parecia cansado. E estressado. "Venha, dê uma volta comigo até a máquina de café. Diga-me o que você acha que é o problema."

Jim *havia* pensado. O dia todo. Mas ele precisava de um café, então concordou. Ele havia sido orgulhoso demais para pedir ajuda; cada vez mais à medida que o dia passava. No entanto, agora, ele percebia que precisava de alguém para ouvir e de um novo ponto de vista. Suas boas ideias haviam se esgotado e ele estava trabalhando no momento com base em palpites e adrenalina.

Jim estava perto demais do problema. Ele havia tentado tudo o que passara inicialmente pela sua cabeça sem ver o quadro geral (ou entendê-

lo). Havia iniciado com uma pré-concepção do problema e não focara em detectar a falha antes de aplicar esparadrapos. Cada um deles havia sido uma "pequena correção" que deveria ter sido a solução, porém o problema simplesmente fora mascarado ou tinha mudado de lugar – era como alisar uma bolha de ar presa no papel de parede.

E Jim havia passado o dia todo fazendo isso. Ele não se sentia nem um pouco mais perto da solução, porém sentia os olhos de todo o restante da equipe atrás de sua cabeça enquanto trabalhava freneticamente em uma correção.

"Não se preocupe", disse Julie. Ela tinha visto isso acontecer várias vezes. Ela mesma já havia feito isso um número suficiente de vezes no passado. E ela sabia muito bem que poderia ser perfeitamente capaz de fazer isso novamente. "Diga-me o que você descobriu até agora." Jim começou a descrever a situação.

Depois de um café e de um bate-papo, Jim se sentiu renovado e tinha um novo foco. À medida que ele explicava o problema completo para Julie, sem que ela dissesse uma única palavra, ocorreu-lhe que ele havia ignorado uma peça enorme do quebra-cabeça. Enquanto descrevia o que iria fazer em seguida, Jim percebeu que ele não tinha visto o verdadeiro problema. Ele descreveu o que iria fazer.

"Isso faz todo o sentido", disse Julie incentivando-o. "Você quer trabalhar comigo?"

"Acho que consigo fazer isso sozinho agora", disse Jim. "Mas volte em dez minutos e verifique se eu não estou perdido novamente." Então ele acrescentou pensativo: "... e quando eu terminar, você poderia revisar a correção?".

"É claro", disse Julie. Ela sorriu.

Jim era parecido com ela em vários aspectos. Ela sabia que ele somente aprenderia cometendo erros. No final do dia, ela pediria a ele que refletisse sobre o que havia acontecido – uma pequena retrospectiva pessoal. Ela torcia para que ele não fizesse isso novamente com pressa.

Jim corrigiu o problema, eles fizeram a revisão da correção e a

disponibilizaram no final do dia (que prosseguiu com uma comemoração regada a bebidas, quando poderia ter sido gasto em cima de um teclado, trabalhando noite adentro).

Desenvolvimento em uma ilha deserta

Nenhum desenvolvedor é uma ilha. Tome cuidado com os perigos de ficar tão excessivamente focado e tão próximo a uma parte do problema de modo a não ser capaz de vê-lo por completo ou não trabalhar nele de maneira eficiente.

Preste atenção em si mesmo. Verifique se você não está indo para um beco de codificação sem saída e certifique-se de que você perceberá e poderá retroceder. Como isso pode ser feito? Identifique alguns mecanismos práticos. Defina limites de tempo curtos e prazos finais próximos e analise o seu progresso à medida que avançar. Faça com que você seja responsável diante de outra pessoa enquanto trabalhar, seja programando aos pares, revisando ou informando-lhe informalmente o seu progresso.

> **PONTO-CHAVE** Seja responsável diante de outro programador. Avalie o seu progresso junto a essa pessoa regularmente.

Jamais seja orgulhoso demais para pedir ajuda. Como acabamos de ver, normalmente é descrevendo um problema que você explicará a si mesmo como corrigi-lo. Se nunca tentou fazer isso, você ficará impressionado com a frequência com que isso acontece. Não é preciso nem mesmo conversar com outro programador. Pode ser até com um patinho de borracha.[1]

Fique na base da montanha

Muitos problemas de design de software, decisões de codificação ou correções de bug parecem ser uma montanha enorme a ser escalada. Correr diretamente até a base da montanha e começar a escalar geralmente é uma abordagem incorreta.

Com frequência, é melhor (ou seja, mais fácil, mais eficiente no que diz respeito ao tempo e ao custo) abordar a montanha com uma equipe. A equipe pode ajudar um ao outro a subir. Uma pessoa pode informar

quando perceber que outra vai subir e ficar em uma situação difícil. A equipe pode trabalhar em conjunto de maneiras que um indivíduo não pode.

Sempre vale a pena parar e planejar um caminho antes de iniciar a sua subida. Ao contornar o lado mais distante da montanha, pode haver um caminho muito mais fácil para subir, se você procurar. De fato, pode até ser que já haja um caminho traçado. Com placas de sinalização. E iluminação. E uma escada rolante. Raramente, o seu primeiro caminho para resolver um problema será o melhor.

> **PONTO-CHAVE** Ao encarar um problema, não se esqueça de considerar mais de uma abordagem para solucioná-lo. Somente então você deverá começar a trabalhar nele.

Esse é um exemplo entre muitos de como o desenvolvimento de software geralmente é mais um problema humano do que técnico. Devemos aprender a solucionar os problemas de maneira mais eficiente e superar nossos instintos naturais de resolvê-los rapidamente – porém de modo ineficiente.

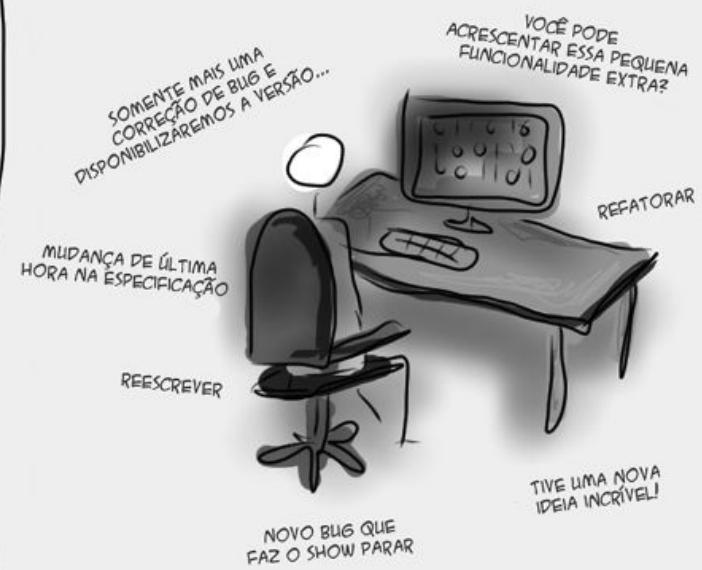
Perguntas

1. Com que nível de eficiência você trabalha com as outras pessoas de sua equipe?
2. Você consegue pedir ajuda ou discutir os problemas?
3. Com que frequência você "codifica até chegar a um beco sem saída"? Quando isso aconteceu pela última vez? Quanto tempo você levou para perceber?
4. Você *é responsável* em relação às outras pessoas? Em caso negativo, em relação a quem você poderia ser responsável?
5. Você acha que compartilhar o seu progresso e discutir os problemas fará você se parecer um programador mais fraco para as outras pessoas da equipe?

Veja também

- *Mais inteligente, e não mais árduo* (Capítulo 31) – Essa história destaca

a importância de trabalhar de forma *mais inteligente, e não mais árdua*.

- *Use o seu cérebro* (Capítulo 17) – Não sofra com a falta de visão, correndo em direção a uma meta pelo caminho errado. Pare e *use o seu cérebro*.
- *Ser responsável* (Capítulo 35) – Ser responsável diante de alguém e ter conversas diárias (ou mais frequentes) sobre o seu trabalho pode ajudar a evitar erros descuidados.

> **TENTE ISTO...** Antes de começar a sua próxima tarefa de codificação, redija um "plano de ataque" sobre como você irá resolver/diagnosticar/projetar/abordar o código. Use isso para evitar que você corra de cabeça em direção à tarefa sem a devida consideração.

---

1 Hunt e Thomas, *The Pragmatic Programmer* (O programador pragmático).

PARTE V
Uma meta de pessoas

O desenvolvimento de software raramente é uma atividade solitária; é um esporte social – uma meta de pessoas. Um bom programador é capaz de trabalhar bem com os outros habitantes da fábrica de software. Para se tornar um programador melhor, você deve aprender a trabalhar de modo eficiente com outras pessoas e a aprender com elas.

CAPÍTULO 34
O poder das pessoas

Duas coisas são infinitas: o universo e a estupidez humana;
e não estou certo sobre o universo.
– Albert Einstein

Programação é uma meta de pessoas.

Praticamente, desde que os primeiros programas foram criados, percebemos que a programação não é somente um desafio técnico. É também um desafio social. O desenvolvimento de software é um passatempo que envolve escrever código *com* outras pessoas *para* outras pessoas entenderem. Significa trabalhar com o código de outras pessoas, entrar e sair de equipes de software, trabalhar sob a supervisão de seu chefe, gerenciar desenvolvedores (que é como pastorear um bando de gatos) e assim por diante.

Muitos dos livros mais duradouros sobre programação são dedicados ao problema das pessoas, por exemplo, *The Mythical Man Month*[1] e *Peopleware*.[2]

Assim como as pessoas que trabalham com uma base de código irão inevitavelmente moldar o código que produzem, as pessoas que trabalharem com você irão inevitavelmente moldar *você*.

PONTO-CHAVE Coloque-se intencionalmente ao lado de excelentes programadores.

Isso quer dizer que, se você quiser ser um programador excepcional, você deverá se colocar, de modo consciente, diariamente entre pessoas que sejam programadores excepcionais. É uma maneira realmente simples, porém profunda, de garantir que você aperfeiçoará suas habilidades e atitudes.

Afinal de contas, somos produtos de nosso ambiente. Assim como as

plantas precisam de um bom solo, fertilizantes e a atmosfera correta para crescerem saudavelmente, o mesmo ocorre conosco.

Passar muito tempo com pessoas deprimentes fará você se sentir deprimido. Passar muito tempo com pessoas exaustas fará você se sentir cansado e letárgico. Passar muito tempo com funcionários desleixados irá estimulá-lo a trabalhar de modo desleixado – por que se importar em tentar se ninguém mais o faz? Por outro lado, trabalhar com indivíduos que sejam apaixonados por um bom código e que se esforcem para criar um software melhor irá estimulá-lo a fazer o mesmo.

Ao ficar imerso em um ambiente com excelentes programadores, você será presenteado com:

- um entusiasmo contagioso;
- uma motivação inspiradora;
- uma responsabilidade contagiante.

Encontre pessoas desse tipo e permaneça em sua companhia. Conscientemente, procure pessoas que se importem com um bom código e em escrevê-lo bem. Nesse tipo de ambiente, você sempre será nutrido e estimulado.

Ao trabalhar com desenvolvedores de grosso calibre, você irá adquirir muito mais do que conhecimento técnico, embora esse conhecimento, por si só, seja bem valioso. Você terá um reforço positivo de hábitos e atitudes para uma boa programação. Será incentivado a crescer e desafiado a melhorar em sua áreas mais deficientes. Isso nem sempre será fácil ou confortável, mas vale a pena.

Portanto tome a decisão de procurar os melhores programadores e de trabalhar com eles. Faça o design do código com eles. Programe com eles. Socialize-se com eles.

O que você deve fazer

Você pode tornar formal esse tipo de relacionamento por meio de um programa de *orientação* (mentorship); de fato, muitos locais bons de

trabalho tentam implantar esquemas de orientação formal na prática. Reserve períodos específicos de tempo para trabalhar em equipe.

Ou você pode tentar isso informalmente: procure trabalhar nos mesmos projetos que os grandes programadores. Mude de emprego para trabalhar com eles. Vá a conferências, palestras ou grupos de usuários para conhecê-los. Ou simplesmente procure sair com outros grandes programadores.

À medida que fizer isso, aprenda com eles. Preste atenção:

- em como eles pensam e resolvem os quebra-cabeças;
- em como eles planejam um caminho em direção aos problemas;
- na atitude que eles adotam quando a situação se torna difícil;
- em como eles sabem se devem continuar pressionando para resolver um problema, quando devem fazer uma pausa ou quando devem tentar uma abordagem diferente;
- em suas habilidades e técnicas específicas de codificação que você ainda não compreende.

Conheça seus experts

Considere cuidadosamente como você imagina ser um programador excelente.

Especificamente, você *não* deve ficar ao lado de pessoas que trabalhem excessivamente, usando todas as horas dadas por Deus codificando. É quase certo que essas pessoas não sejam programadores excepcionais! Os gerentes normalmente acham que os funcionários que passam todas as horas do dia em um projeto são os heróis da programação, porém, com frequência, isso realmente é um indício de sua falta de capacidade. Eles não conseguem fazer a tarefa certa da primeira vez, portanto precisam gastar muito mais horas para fazer o código "funcionar" do que é realmente necessário.

Os experts fazem com que pareça fácil e concluem uma tarefa no prazo.

Campo de visão 20/20

À medida que olho para trás e vejo minha carreira, percebo que os momentos mais agradáveis e pessoalmente produtivos que tive foram aqueles em que trabalhei junto com desenvolvedores excelentes, motivados e interessantes. E por causa disso, atualmente, sempre tento me colocar junto a pessoas como essas.

Aprendi que elas me fazem ser melhor no que faço e que me divirto mais enquanto o faço.

Um efeito colateral interessante e benéfico de trabalhar com bons programadores é que é muito mais provável que você acabe trabalhando com um código bom.

Perguntas

1. Você está cercado de pessoas as quais você considera programadores excelentes no momento? Por quê? Ou por que não?
2. Como você pode se aproximar de programadores melhores? Você pode mudar-se para um novo projeto ou entrar em uma nova equipe? Você acha que é hora de procurar outro emprego em uma empresa diferente?
3. Como você pode dizer quem é um desenvolvedor excelente e quem não é?

Veja também

- *Aprecie o desafio* (Capítulo 26) – Procure bons colegas que possam incentivá-lo e desafiá-lo.
- *Ser responsável* (Capítulo 35) – Seja responsável diante dos outros.
- *Fale!* (Capítulo 36) – Aprenda a se comunicar bem; ouvir é essencial se você pretende aprender.
- *Ode ao código* (Capítulo 38) – Nem todo colega é santo.

TENTE ISTO... Identifique alguns "heróis da codificação" com quem você gostaria de aprender e planeje uma maneira de trabalhar com eles. Considere pedir que você seja orientado por eles.

10.000 MACACOS

(OU ALGO POR AÍ)

O PODER DAS PESSOAS

DESENVOLVIMENTO DE SOFTWARE É UMA META DE PESSOAS

MODELOS PROBLEMÁTICOS DE TRABALHO EM EQUIPE RESULTAM EM UM SOFTWARE RUIM

O CABO-DE-GUERRA

VOCÊ DEVE GARANTIR QUE HAJA RESPEITO MÚTUO E PROMOVER UMA COOPERAÇÃO SADIA

DIGO QUE DEVEMOS DISPONIBILIZAR A VERSÃO, MEU CARO

NÃO SENHOR; ELA AINDA NÃO ESTÁ PRONTA

A LUTA

AS EQUIPES NÃO DEVEM SER ADVERSÁRIAS, MAS DEVEM TRABALHAR JUNTAS PARA O BEM DO PROJETO

HUMMM, PARECE SER UM CÓDIGO SABOROSO

VOU DAR UMA MORDIDA

PROGRAMAÇÃO "PEAR"

LEMBRE-SE: AS PESSOAS TRABALHAM MELHOR JUNTAS QUANDO COMEM JUNTAS

---

1 Frederick P. Brooks Jr., *The Mythical Man Month* (Boston: Addison Wesley, 1995).
2 Tom Demarco e Timothy Lister, *Peopleware* (Nova York: Dorset House Publishing, 1999).

CAPÍTULO 35

Ser responsável

Pensar bem é sábio; planejar bem é mais sábio; fazer bem é o mais sábio e melhor.

– Provérbio persa

Eu corro. Toda semana. É por causa da minha cintura. Talvez seja culpa, mas eu sinto que devo fazer algo para mantê-la sob controle.

Mas sejamos claros: não sou masoquista. Fazer exercícios *não* é a atividade de que mais gosto no mundo. Longe disso. Ela está um pouco acima de espetar ferros quentes nos meus olhos. Há muitas outras atividades que eu preferiria fazer em minhas noites. Muitas delas envolvem ficar sentado, de preferência com uma taça de vinho.

Entretanto eu sei que *devo* correr. Faz bem para mim.

Esse fato, por si só, é suficiente para garantir que eu vá regularmente, todas as semanas, e faça um treino completo? Sem ser desleixado ou reduzir o ritmo?

Não.

Não gosto de fazer exercícios e empregaria alegremente a menor das desculpas para escapar de uma corrida. "Ah, não, meu short de corrida está descosturado." "Ah, não, meu nariz está escorrendo." "Ah, não, estou um pouco cansado." "Ah, não, minha perna caiu."

(Tudo bem, algumas desculpas *são* melhores que outras.)

Que forças ocultas me impelem a continuar correndo regularmente se somente a culpa não consegue me arrastar porta afora? Que poder mágico me empurra quando a força de vontade falha?

Responsabilidade.

Eu corro com um amigo. Essa pessoa sabe quando estou sendo desleixado

e me incentiva a sair de casa mesmo quando não estou com vontade. Essa pessoa aparece em minha porta, conforme combinado, antes que minha letargia se instale. Eu lhe presto o mesmo serviço. Já perdi a conta das vezes que eu não teria corrido ou teria desistido no meio do caminho se eu não tivesse alguém lá, observando-me e correndo comigo.

E, como subproduto, gostamos de correr mais pela companhia e pela experiência compartilhada.

Às vezes, *ambos* não estamos dispostos a correr. Mesmo admitindo isso um para o outro, não deixamos o outro na mão. Nós nos incentivamos a superar a dor. E depois que corremos, sempre ficamos felizes por termos feito isso, mesmo que na hora não tenha parecido ser uma ótima ideia.

Alongando a metáfora

Algumas metáforas são dispositivos literários tênues, criados para entreter ou para serem usados como ideias artificiais. Algumas são tão indiretas que chegam a provocar distrações, ou formam um paralelo ruim a ponto de serem totalmente enganosas.

Entretanto acredito que essa imagem de responsabilidade é diretamente relevante para a qualidade de nosso código.

Ouvimos os experts de nosso mercado, os palestrantes, os escritores e os profetas de código falarem sobre produzir um bom código bem escrito. Eles enaltecem as virtudes de um código "limpo" e explicam por que precisamos de um código bem fatorado. Mas de nada importaria se, no calor do ambiente de trabalho, não pudéssemos colocar isso em prática. Se as pressões da codificação nos fizessem deixar de lado a nossa moral de desenvolvimento e passássemos a codificar desleixadamente como idiotas desinformados, qual seria a utilidade de seus conselhos?

O espírito está disposto; porém, quando os prazos finais se assomam com muita frequência, a carne é fraca. Podemos reclamar do estado precário de nossas bases de código, mas a quem devemos culpar?

Devemos incluir maneiras de evitar a tentação de usar atalhos, remendos e correções rápidas em nosso regime de desenvolvimento. Devemos ter

algo que nos atraia para longe da armadilha do design sem planejamento, das soluções desleixadas e fáceis e das práticas pela metade. É o tipo de tarefa que exige esforço, mas, quando olhamos para trás, ficamos felizes de *termos* feito.

Como você acha que isso pode ser alcançado?

A responsabilidade ajuda

Eu sei que, em minha carreira até agora, o único aspecto mais importante que tem me incentivado a trabalhar da melhor maneira permitida pelas minhas habilidades é a *responsabilidade* diante de uma equipe de ótimos programadores.

São os outros programadores que me fazem parecer bem. São esses outros programadores que me *tornaram* um programador melhor.

> **PONTO-CHAVE** Ser responsável pela qualidade de seu trabalho diante de outros programadores irá melhorar drasticamente a qualidade de sua codificação.

É uma ideia simples, porém eficaz.

Código++

Para garantir que está escrevendo um código excelente, você precisa de pessoas que o verifiquem a cada passo do caminho. Pessoas que garantirão que você está fazendo o melhor que suas habilidades permitem e que está mantendo o padrão de qualidade do projeto em que estiver trabalhando.[1]

Esse não precisa ser um processo burocrático vigiado nem um plano de desenvolvimento pessoal militar que tenha influência direta em seu salário. Com efeito, é melhor que não seja. Um sistema de responsabilidade leve, sem muitas formalidades, que não envolva formulários, sessões longas de revisão ou revisões formais é muito mais adequado e terá resultados muito melhores.

Acima de tudo, é importante simplesmente reconhecer a necessidade da

responsabilidade; responder às outras pessoas pela qualidade de seu código estimula você a trabalhar da melhor maneira possível. Perceba que colocar-se ativamente na posição vulnerável de ser responsável não é um sinal de fraqueza, porém uma maneira valiosa de receber feedback e de melhorar suas habilidades.

Que nível de responsabilidade você acha que tem atualmente pela qualidade do código que você produz? Alguém o está desafiando a produzir um trabalho de alta qualidade para evitar que você se deixe levar por práticas ruins e desleixadas?

Vale a pena ter responsabilidade como meta não só para a qualidade do código produzido, mas também para a maneira como aprendemos e como planejamos o nosso desenvolvimento pessoal. É benéfico até mesmo quando se trata de personalidade e de vida pessoal (mas isso é assunto para outro livro).

Fazendo com que dê certo

Há algumas maneiras simples de atribuir responsabilidade pela qualidade do código em seu processo de desenvolvimento. Em uma equipe de desenvolvimento, percebemos que foi particularmente produtivo quando todos os programadores concordaram com uma simples regra: *todo código deve passar por dois pares de olhos antes de ser inserido no sistema de controle de versões*. Sendo essa uma regra acordada com um parceiro, era *nossa escolha* ser responsável um diante do outro, em vez de ser responsabilidade de algum gerente ditador de cima, com seus ternos e faces anônimas. Um comprometimento profundo foi a chave do sucesso desse esquema.

Para satisfazer à regra, empregamos programação aos pares e/ou uma revisão de código um a um, sem muitas formalidades, mantendo cada check-in de alteração pequeno para permitir que o esquema fosse administrável. Saber que outra pessoa iria examinar o seu trabalho era suficiente para promover uma resistência a uma prática desleixada e melhorar a qualidade geral do código.

PONTO-CHAVE Se você souber que outra pessoa *irá* ler e comentar o seu código, é mais provável que você vá escrever um código bom.

Essa prática realmente melhorou a qualidade da equipe também. Todos nós aprendemos uns com os outros e compartilhamos os nossos conhecimentos sobre o sistema. Ela nos estimulou a sermos mais responsáveis pelo sistema e a compreendê-lo melhor.

Acabamos também cooperando mais de perto, apreciamos trabalhar uns com os outros e nos divertimos mais escrevendo o código, como consequência desse esquema. A responsabilidade resultou em um fluxo de trabalho mais agradável e mais produtivo.

Definindo o padrão

Ao fazer com que o desenvolvedor seja responsável em sua rotina diária, vale a pena investir tempo para considerar o benchmark que você tem como meta. Faça a si mesmo as seguintes perguntas:

Como a qualidade de seu trabalho é julgada? Como as pessoas *atualmente* classificam o seu desempenho? Qual é o critério usado para mensurar a sua qualidade? Como você acha que elas *deveriam* classificá-lo?

- O software funciona, isso é bom o suficiente.
- Foi escrito rapidamente e disponibilizado no prazo (a qualidade interna não é o mais importante).
- Foi bem escrito e pode ser facilmente mantido no futuro.
- Alguma combinação das opções anteriores.

Qual das opções é vista como a mais importante?

Quem julga o seu trabalho atualmente? Quem é o público-alvo de seu trabalho? Ele é visto somente por você? Por seus pares? Pelos seus superiores? Pelo seu gerente? Pelo seu cliente? De que modo eles são qualificados para julgar a qualidade de seu trabalho?

Quem *deve* ser o árbitro da qualidade de seu trabalho? Quem realmente sabe como foi o seu desempenho? Como você pode fazer com que essas

pessoas sejam envolvidas? É tão simples quando perguntar a elas? A opinião delas tem alguma influência na visão atual que a empresa tem da qualidade de seu trabalho?

Sobre quais aspectos de seu trabalho você deve ser responsável diante dos outros?

- As linhas de código que você produz?
- O design?
- A conduta e o processo que você usou para desenvolvê-lo?
- A maneira como você trabalhou com outras pessoas?
- As roupas que você vestiu ao fazê-lo?

Quais aspectos são os mais importantes para você neste momento? Em relação a quais aspectos você deve ter mais responsabilidade e deve receber mais incentivo para continuar melhorando?

Os próximos passos

Se você acha que isso é importante, e é algo que você deva começar a adicionar ao seu trabalho:

- Concorde que a responsabilidade é algo bom. Comprometa-se com ela.
- Encontre alguém a quem você deva responsabilidade. Considere fazer disso um acordo recíproco; talvez você possa envolver toda a equipe de desenvolvimento.
- Considere a implementação de um esquema simples como aquele descrito anteriormente para a sua equipe, em que toda linha de código alterada, acrescentada ou removida deva passar por dois pares de olhos.
- Chegue a um consenso sobre como você tratará a responsabilidade – pequenas reuniões, revisões no final da semana, reuniões de design, programação aos pares, revisões de código etc.
- Comprometa-se com um determinado nível de qualidade do trabalho; esteja preparado para ser desafiado quanto a isso. Não fique na

defensiva.

- Se isso abranger toda a equipe ou todo o projeto, garanta que todos estejam comprometidos. Esboce um conjunto de padrões para a equipe ou um código de conduta para o grupo, relacionados à qualidade do desenvolvimento.

Além disso, considere abordar a questão pelo outro lado: você pode ajudar outra pessoa com feedbacks, incentivos e responsabilidade? Você pode se tornar uma referência moral de software para outro programador?

Geralmente, esse tipo de responsabilidade funciona melhor aos pares, e não em um relacionamento de subordinação.

Conclusão

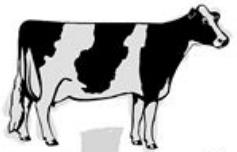
A responsabilidade entre programadores exige certo grau de coragem; você deve estar disposto a aceitar críticas. E deve ter tato o suficiente para fazê-las. Porém as vantagens podem ser marcantes e profundas na qualidade do código que você criar.

Perguntas

1. Até que ponto você é responsável diante dos outros pela qualidade de seu trabalho?
2. Em relação a quê você deve ser responsável diante dos outros?
3. Como você pode garantir que o trabalho que faz hoje é tão bom quanto o trabalho anterior?
4. Como o seu trabalho atual está ensinando-o e está ajudando-o a se aperfeiçoar?
5. Quando você se sentiu feliz por ter mantido a qualidade, mesmo que não tenha achado que fez isso?
6. A responsabilidade funciona somente quando você *opta* por entrar em um relacionamento de responsabilidade, ou ela pode ser algo que você seja *obrigado* a fazer?

Veja também

- *O programador ético* (Capítulo 28) – Você deve ser responsável pela qualidade de sua conduta bem como pela qualidade de seu código.
- *O poder das pessoas* (Capítulo 34) – Ao trabalhar com programadores excelentes, é impossível não ser desafiado para melhorar suas habilidades.
- *Dessa vez, eu consigo* (Capítulo 33) – Ser responsável diante de outros pode ajudar a evitar embaraços e erros tolos.
- *Jogando segundo as regras* (Capítulo 15) – Empregue a responsabilidade entre os membros da equipe para incentivar todos a obedecerem as “regras” de sua equipe.

TENTE ISTO... Encontre um colega diante de quem você deverá ser responsável. Comprometa-se com determinado nível de qualidade no trabalho. Peça que o seu colega fique de olho em seu código. Considere fazer desse um relacionamento bidirecional.

---

1 Esse é um dos motivos pelos quais um código aberto normalmente tem mais qualidade do que um código proprietário: você sabe que vários outros programadores estarão vendo o seu trabalho.

CAPÍTULO 36

Fale!

O único grande problema da comunicação é a ilusão de que ela ocorreu.

– George Bernard Shaw

É o estereótipo clássico de um programador: um geek antissocial que trabalha sozinho como um escravo, em uma sala apertada com pouca luz, debruçado em um console, digitando furiosamente. Jamais vê a luz do dia. Jamais fala com outra pessoa "da vida real".

Porém nada poderia estar mais distante da verdade.

Esse trabalho tem *tudo* a ver com comunicação. Não é exagero dizer que somos bem-sucedidos ou não de acordo com a qualidade de nossa comunicação.

Essa comunicação vai além das conversas que iniciamos no bebedouro. Apesar de elas serem essenciais. Vai além das conversas na lanchonete, no almoço ou no bar. Apesar de todas elas também serem essenciais.

A nossa comunicação é muito mais profunda; ela é multifacetada.

Código é comunicação

O software em si – o ato propriamente dito de escrever código – é uma forma de comunicação.

Isso funciona de diversas maneiras...

Falando com as máquinas

Quando escrevemos código, estamos *conversando* com o computador por meio de um interpretador. Esse pode ser literalmente um "interpretador" para linguagens de scripting, interpretadas em tempo de execução. Ou

podemos nos comunicar por meio de um tradutor: um compilador ou JIT (Just-In-Time). Poucos programadores atualmente conversam usando a linguagem natural da CPU: o código de máquina.

Nosso código existe para fornecer uma lista literal de instruções para a CPU.

Com muita frequência, minha esposa me deixa uma lista de tarefas para fazer. *Faça o jantar, limpe a sala de estar, lave o carro.* Se suas instruções forem ilegíveis ou não estiverem claras, não farei o que ela realmente quer que eu faça. Irei passar ferro nas facas e passar aspirador de pó na banheira. (Aprendi a não discutir e a fazer o que me foi dito, mesmo que não faça sentido para mim.) Se ela quiser ter os resultados adequados, ela deverá me deixar os tipos certos de instruções.

O mesmo ocorre com o nosso código.

Programadores desleixados não são explícitos. Os resultados de seus códigos podem ser o equivalente a passar ferro nas facas.

> **PONTO-CHAVE** O código representa uma comunicação com o computador. Ele deve ser claro e não deve ser ambíguo para que suas instruções sejam executadas conforme pretendido.

Não estamos falando na língua-mãe da CPU; sendo assim, é sempre importante saber quais nuances dessa linguagem são perdidas na tradução de nossa linguagem de programação. A conveniência de usar a nossa linguagem preferida tem um custo.

Falando com os animais

Embora o seu código forme uma conversação contínua com o seu amigo mecânico – o computador –, ele não fala *somente* com uma CPU.

Ele fala com outros seres humanos também – outras pessoas que compartilham o código com você e que deverão *ler* o que você escreveu. O código será lido pelas pessoas com quem você estiver trabalhando. Será lido pelas pessoas que revisarem o seu trabalho. Será lido pelo programador responsável pela manutenção, que verá o seu código posteriormente. Será lido por você quando você retornar depois de alguns

meses para corrigir bugs desagradáveis em seu antigo trabalho.

> **PONTO-CHAVE** O seu código representa uma comunicação com outros seres humanos. Inclusive com você. Ele deve ser claro e não pode ser ambíguo para o caso de outras pessoas darem manutenção nele.

Isso é importante.

Um programador de grosso calibre se esforça para criar um código que comunique claramente a sua intenção. O código deve ser transparente: deve expor os algoritmos sem obscurecer a lógica. Ele deve permitir que outras pessoas o modifiquem facilmente.

Se o código não se revelar, mostrando o que faz, será difícil alterá-lo. E um fato que sabemos sobre codificação no mundo real é que *a única constante é a mudança*. Um código que não se comunique será um gargalo e impedirá desenvolvimentos futuros.

Um bom código não deve ser conciso a ponto de ser ilegível. Porém não deve ser extenso nem trabalhoso. E, definitivamente, não deverá estar cheio de comentários. Mais comentários *não* tornam o código melhor; eles somente o tornam mais extenso – e provavelmente pior, pois os comentários podem facilmente deixar de estar sincronizados com o código.

> **PONTO-CHAVE** Mais comentários *não* tornam necessariamente o seu código melhor. Um código comunicativo não precisa de comentários extras para ser adequado.

Um bom código não é ardilosamente inteligente, a ponto de usar engenhosamente os recursos “avançados” da linguagem com tanta pose que os programadores responsáveis pela manutenção ficarão coçando a cabeça. (É claro que a quantidade de cabeças coçando depende da qualidade dos programadores responsáveis pela manutenção; esse tipo de situação sempre depende do contexto.)

A qualidade de nossa expressão no código é determinada pelas linguagens de programação que escolhemos usar e pela forma como as utilizamos. Você está usando uma linguagem que permita expressar naturalmente os conceitos sendo modelados?

Devemos usar a mesma linguagem ao mesmo tempo, ou sofreremos de uma cacofonia bíblica como a da Torre de Babel. A equipe que estiver trabalhando em uma seção de código deve usar a mesma linguagem; adicionar linhas de Basic em um script Python não é uma fórmula vencedora. Se a sua aplicação toda estiver escrita em C++, a primeira pessoa a adicionar um código em outra linguagem deverá ter um bom motivo para isso.

Entretanto, mesmo em um ambiente em que a mesma linguagem de programação seja utilizada, é possível usar diferentes dialetos e acabar introduzindo barreiras de comunicação. Diferentes convenções de formatação podem ser adotadas ou diferentes idioms de codificação (por exemplo, usar C++ "moderno" *versus* "C++ como um C melhorado") podem ser empregados.

É claro que usar várias linguagens de programação não é algo ruim. Projetos maiores podem ser legitimamente compostos de códigos em mais de uma linguagem. Isso é padrão para sistemas distribuídos de grande porte, em que o backend é executado em um servidor em uma linguagem, enquanto clientes remotos são implementados em outras linguagens, geralmente mais dinâmicas, hospedadas em navegadores. Esse tipo de arquitetura permite empregar o tipo certo de linguagem para cada tarefa. Vemos, nesse caso, outra linguagem em jogo: a linguagem com que essas partes se comunicam (pode ser uma API REST com formatação de dados JSON).

Considere também a *linguagem natural* em que você programa. A maioria das equipes está localizada no mesmo país, portanto isso não é um problema. Entretanto, com frequência, eu trabalho em projetos envolvendo vários países, com muitos falantes não nativos de inglês. Fizemos uma opção consciente no sentido de escrever todo o código em inglês: todos os nomes de variáveis, os comentários, os nomes de classes e de funções, enfim, tudo. Isso nos possibilita certo grau de sanidade.

Já trabalhei em projetos multisite que não faziam isso, e ter de passar os comentários do código pelo Google Translate para descobrir se eles são ou não importantes é um verdadeiro problema. Já fiquei me perguntando

se um nome de variável estava em húngaro, continha um erro ortográfico, estava abreviado ou se eu tinha um domínio muito ruim da linguagem natural que usava.

> **PONTO-CHAVE** O modo como um código se comunica depende da linguagem de programação, dos idioms empregados e da linguagem natural subjacente. Todos esses fatores devem ser compreendidos para que a leitura possa ser efetuada.

Lembre-se de que o código é lido por seres humanos com muito mais frequência do que é escrito. Desse modo, ele deverá estar otimizado para a leitura, e não para a escrita. Use uma construção concisa somente se for mais fácil para outra pessoa entender, em vez de ser mais fácil para você digitar. Siga uma convenção de layout que revele claramente a intenção, e não uma que exija poucas teclas.

Falando com ferramentas

O nosso código se comunica também com outras ferramentas que trabalhem com ele. Nesse caso, “ferramentas” não é um eufemismo para os seus colegas.

O seu código pode ser fornecido a geradores de documentação, sistema de controle de versões, software de monitoração de bugs e analisadores de código. Até mesmo os editores que usamos podem influenciar (que codificação de conjunto de caracteres o seu editor está usando?).

Não é incomum adicionar diretivas extras em nosso código para saciar as exigências desses processadores ou adaptar o nosso código para que fique mais adequado a essas ferramentas (ajustando a formatação, o estilo de comentários ou os idioms de codificação).

Como isso afeta a legibilidade do código?

Comunicação interpessoal

A comunicação elétrica jamais será uma substituta para o rosto de alguém que, com sua alma, estimule outra pessoa a ser corajosa e verdadeira.

– Charles Dickens

Nós não nos comunicamos simplesmente digitando código. Os programadores trabalham em equipe junto a outros programadores. E com uma organização mais ampla.

Há muita comunicação ocorrendo nesse caso. Como fazemos isso o tempo todo, os programadores de alta qualidade *devem* ter uma comunicação de alta qualidade. Escrevemos mensagens, até mesmo gesticulamos, para falar com outras pessoas o tempo todo.

Maneiras de conversar

Há vários canais de comunicação que usamos para conversar, notadamente:

- conversa face a face;
- conversa por telefone, um a um;
- conversa por telefone usando "conference call";
- conversa por meio de canais de VoIP (que não é necessariamente diferente do telefone, porém é mais provável que deixe as mãos livres e permita enviar arquivos pelo mesmo canal de comunicação);
- email;
- mensagens instantâneas (por exemplo, digitar no Skype ou usar canais IRC, salas de bate-papo ou SMS);
- videoconferências;
- envio de cartas escritas por meio do sistema postal físico (você se lembra dessa prática bizarra?);
- fax (que foi amplamente substituído pelos scanners e pelo bom senso; entretanto ele ainda tem lugar em nosso panteão da comunicação, pois é considerado útil para envio de documentos de caráter legal).

Cada um desses sistemas é diferente, variando quanto à área que abrange, à quantidade de pessoas envolvidas em cada extremidade da comunicação, aos recursos disponíveis e à riqueza da interação (a outra pessoa pode ouvir o tom de sua voz ou ler a sua linguagem corporal?), à duração típica, à urgência necessária e à possibilidade de adiar uma discussão e à maneira como uma conversação é iniciada (por exemplo, é

preciso que uma solicitação de reunião seja feita ou é aceitável interromper alguém sem aviso prévio?).

Cada opção tem etiquetas e convenções diferentes e exige habilidades diferentes para ser usada de modo eficiente. É importante selecionar o canal correto de comunicação para a conversa que deve ocorrer. Qual é a urgência da resposta? Quantas pessoas devem estar envolvidas?

Não envie um email a alguém quando precisar de uma resposta urgente; o email pode ser ignorado por dias. Vá até onde a pessoa estiver, telefone, faça contato pelo Skype. Por outro lado, não telefone se o problema não for urgente. O tempo das outras pessoas é precioso e a sua interrupção irá desestruturar o *fluxo* de trabalho delas, impedindo que elas trabalhem em suas tarefas atuais.

Quando precisar fazer uma pergunta a alguém, considere se você está prestes a usar o sistema de comunicação correto.

> **PONTO-CHAVE** Domine as diferentes formas de se comunicar. Utilize o sistema apropriado para cada conversa.

Observe o seu linguajar

À medida que um projeto evolui, ele adquire o seu próprio dialeto: um vocabulário do projeto e termos específicos do domínio, além dos idioms predominantes, usados no design ou para pensar no formato do design de software. Também definimos a terminologia do processo usado para trabalharmos juntos (por exemplo, falamos de *histórias de usuário*, *épicos*[1], *sprints*[2]).

> **PONTO-CHAVE** Tome cuidado para usar o vocabulário correto com as pessoas corretas.

O seu cliente deve ser forçado a aprender termos técnicos? O seu CEO precisa conhecer a terminologia do desenvolvimento de software?

Linguagem corporal

Você ficaria chateado se alguém se sentasse ao seu lado, iniciasse uma conversa, porém ficasse o tempo todo com o rosto voltado para o outro

lado. (Ou você poderia fingir que a pessoa saiu de um filme ruim de espiões: *ouvi falar que este ano as groselhas estão produzindo bem... e o mesmo acontece com as mangas*.[3])

Se a pessoa fizesse cara feia sempre que você falasse, você ficaria ofendido. Se ela ficasse brincando com um cubo mágico durante a conversa, você se sentiria desvalorizado.

É fácil fazer exatamente isso quando nos comunicamos eletronicamente, ou seja, não respeitar totalmente a pessoa com quem estamos conversando. Em uma conversa exclusivamente por meio de voz, é fácil se desviar, ler emails, navegar pela Web e não prestar atenção totalmente em alguém.

Tendo abraçado completamente a era moderna da banda larga sempre conectada, agora prefiro optar por um canal de comunicação com *vídeo*. Geralmente, eu inicio uma conversa que poderia ter sido feita por telefone ou por mensagem instantânea usando um bate-papo com câmera por meio de VoIP. Mesmo que meu parceiro de conversa não habilite a sua câmera, gosto de enviar uma imagem para que meu rosto e a linguagem corporal estejam claramente visíveis.

Isso mostra que não estou escondendo nada e promove uma conversa mais aberta.

Um bate-papo com vídeo força você a se concentrar na conversa. Ela envolve mais intensamente a outra pessoa e mantém o foco.

Comunicação paralela

O seu computador participa de várias conversas ao mesmo tempo: fala com o sistema operacional, com outros programas, com device drivers e com outros computadores. Ele é realmente muito inteligente. Devemos garantir que a comunicação de *nosso* código com ele seja clara e que não crie confusão enquanto ele estiver conversando com outros códigos.

Essa é uma analogia eficaz com a nossa comunicação interpessoal. Com tantos canais de comunicação simultaneamente disponíveis, poderíamos estar envolvidos em uma brincadeira no escritório, falando por meio de mensagens instantâneas com um funcionário remoto e trocando SMSs

com nosso(a) parceiro(a), tudo isso enquanto participamos de várias sequências de emails.

E então o telefone toca. Toda a sua pilha cambaleante de comunicação cai.

Como você pode garantir que cada uma de suas conversas seja clara e bem estruturada o suficiente para não confundir qualquer outra comunicação em que você esteja envolvido no momento?

Já perdi a conta da quantidade de vezes que digitei a resposta errada na janela errada do Skype e deixei alguém confuso. Felizmente, jamais revelei informações confidenciais da empresa dessa maneira. Ainda.

PONTO-CHAVE Uma comunicação eficiente exige foco.

Conversas nas equipes

A comunicação é o óleo que lubrifica o trabalho em equipe. É simplesmente *impossível* trabalhar com outras pessoas e não conversar com elas.

Isso, mais uma vez, ressalta a lei de Conway. O seu código se molda em torno da estrutura de comunicações entre suas equipes. As fronteiras de suas equipes e a eficiência de suas interações moldam e são moldadas pelo modo como elas se comunicam.

PONTO-CHAVE Uma boa comunicação promove um bom código. A forma como você se comunica moldará o seu código.

Comunicações saudáveis promovem camaradagem e fazem de seu local de trabalho um lugar agradável para viver. Comunicações que não sejam saudáveis rapidamente acabam com a confiança e criam obstáculos para um trabalho em equipe. Para evitar isso, você deve conversar com as pessoas com respeito, confiança, amizade, preocupação, sem ter motivos ocultos nem mostrar agressividade.

PONTO-CHAVE Fale com os outros de modo transparente, com uma atitude saudável, de modo a promover um trabalho eficiente em equipe.

A comunicação em uma equipe deve fluir livremente e ser frequente. Deve

ser normal compartilhar informações, e a voz de todos deve ser ouvida.

Se as equipes não conversarem frequentemente, se elas deixarem de compartilhar seus planos e seus designs, as consequências inevitáveis serão duplicação de código e de esforço. Veremos designs conflitantes na base de código. Haverá falhas quando os trabalhos forem integrados.

Muitos processos estimulam uma comunicação específica e estruturada, com uma cadência definida; quanto mais frequente, melhor. Algumas equipes têm uma reunião semanal para verificar o progresso, mas isso não é realmente bom o suficiente. Reuniões *diárias* e breves são muito melhores (geralmente são chamadas de *scrums* ou de *stand-up meetings*, ou seja, reuniões de pé). Essas reuniões ajudam a compartilhar o progresso, levantar problemas e identificar obstáculos sem a atribuição de culpa. Elas garantem que todos tenham uma visão clara do estado atual do projeto.

O truque nessas reuniões é mantê-las breves e objetivas; se não houver cuidado, elas se degradarão e passarão a ser discussões desconexas e enfadonhas de problemas irrelevantes. Fazer com que suas durações não sejam excedidas também é importante. Do contrário, elas podem se transformar em distrações que interromperão o seu *fluxo* de trabalho.

Falando com o cliente

Há várias outras pessoas com quem devemos conversar para desenvolver um software excelente. Uma das conversas mais importantes que devemos ter é com o cliente.

Devemos entender o que ele quer; caso contrário, não poderemos efetuar a implementação. Portanto você deve perguntar e trabalhar na linguagem dele para definir seus requisitos.

Depois de ter perguntado uma vez, é fundamental continuar conversando com o cliente enquanto você prossegue para garantir que o que ele quer não mudou e que as pressuposições feitas atendem às suas expectativas.

A única maneira de fazer isso é usar a linguagem dele (e não a sua) e utilizar vários exemplos que ele compreenda – por exemplo, demos do

sistema em construção.

Outras formas de comunicação

E, apesar disso, a comunicação do programador vai muito além do que foi descrito. Nós não simplesmente escrevemos código nem simplesmente conversamos. O programador se comunica de outras maneiras, por exemplo, escrevendo documentações e especificações, publicando artigos em blogs ou escrevendo para publicações técnicas.

De quantas maneiras você está se comunicando como programador?

Conclusão

Primeiro aprenda o significado do que você diz e depois fale.

– Epiteto

Um bom programador tem boas habilidades de comunicação como marca registrada. Uma comunicação eficiente:

- é clara;
- é frequente;
- é respeitosa;
- é realizada nos níveis corretos;
- utiliza o meio correto.

Devemos estar atentos a isso e *praticar* a comunicação – devemos procurar melhorar constantemente a comunicação escrita, verbal e por meio de código.

Perguntas

1. Como o tipo de personalidade afeta as suas habilidades de comunicação? Como um programador introvertido pode se comunicar de modo mais eficiente?
2. Até que ponto nossas interações devem ser formais ou casuais? Isso depende do meio de comunicação?
3. Como você pode manter os colegas a par de seu trabalho sem

incomodá-los constantemente com isso?

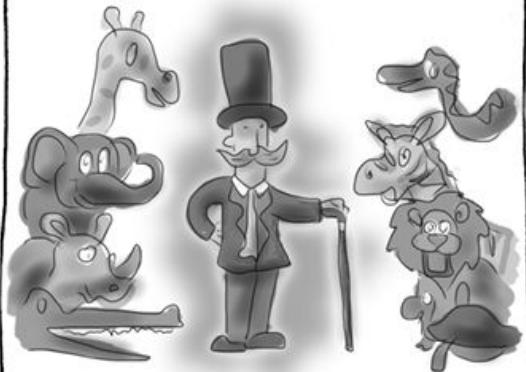
4. De que forma uma comunicação com um gerente difere da comunicação com um colega programador?

5. Que tipo de comunicação é importante para garantir que um projeto de desenvolvimento transcorra de forma bem-sucedida?

6. Qual é a melhor maneira de comunicar um design de código? Dizem que uma imagem vale por mil palavras. Isso é verdade?

7. Equipes distribuídas devem interagir e se comunicar *mais* do que equipes que estejam no mesmo lugar?

8. Quais são as barreiras mais comuns para uma comunicação eficiente?

Veja também

- *Mantendo as aparências* (Capítulo 2) – Código é comunicação. Esse capítulo mostra como se comunicar de modo eficiente no código.
- *Mais inteligente, e não mais árduo* (Capítulo 31) – Manter-se em comunicação constante com a sua equipe, o gerente ou o cliente pode evitar que você trabalhe acidentalmente na atividade incorreta. É bom conversar!

TENTE ISTO... Durante a próxima semana, observe todas as maneiras pelas quais você se comunica com outras pessoas. Determine duas atitudes práticas que você possa adotar para melhorar a qualidade de sua comunicação.

10.000 MACACOS

(OU ALGO POR AÍ)

SOBRE A COMUNICAÇÃO

DOUTOR DOLITTLE

ELE FALAVA COM OS ANIMAIS

DOUTOR LITTLECÓDIGO

TENTOU CONVERSAR COM OS GEEKS.
ACHOU-OS ENFADONHOS.
VOLTOU PARA O SEU COMPUTADOR.

---

1 N.T.: Um épico é uma coleção de histórias de usuário.

2 N.T.: Um sprint é a unidade básica de desenvolvimento em Scrum. Sprints tendem a durar entre uma semana e um mês e são um esforço dentro de uma faixa de tempo (ou seja, restrito a uma duração específica) de comprimento constante. (*fonte: http://pt.wikipedia.org/wiki/Scrum*)

3 Veja o esquete “Secret Service Dentists” (Dentistas do serviço secreto) de Flying Circus (Circo voador) do Monty Python.

CAPÍTULO 37

Manifestos

Confusão de objetivos e perfeição de meios, em minha opinião, parecem caracterizar a nossa era.

– Albert Einstein

Está se tornando uma epidemia! Eles estão surgindo em todos os lugares. Estão saindo pelos nossos ouvidos. É como se você não pudesse escrever uma linha de código, iniciar um projeto ou nem mesmo pensar em desenvolvimento de software sem que seja preciso assinar um.

Os manifestos estão em *todos os lugares*.

Com todos esses diferentes manifestos para desenvolvimento de software, nossa profissão corre o risco de ser mais sobre política do que sobre a arte, o trabalho manual, a ciência e o comércio do desenvolvimento de software.

É claro que o desenvolvimento de software profissional é notadamente um *problema de pessoas*. Portanto a atividade necessariamente envolve determinado volume de política. Porém estamos transformando até mesmo os princípios fundamentais de codificação em uma batalha política.

Alguns dos manifestos de desenvolvedores são gloriosamente ambíguos; parecem mais com um horóscopo do desenvolvimento. E, infelizmente, quando um manifesto se torna popular, vemos facções se formarem em torno dele, levando a disputas sobre o que o manifesto realmente quer dizer. Debates inteiros surgem em torno da exegese de itens particulares

de um manifesto.

A religião do software está viva e passa bem.

Os manifestos parecem surgir para qualquer propósito concebível. Mas eu tenho uma solução. Para conter o fluxo e facilitar a vida dos futuros ativistas de software que queiram criar um manifesto próprio, apresento aqui um manifesto único, abrangente e genérico de desenvolvimento de software. O `Manifesto<ASSUNTO_PREFERIDO>`, se você me permite.

Um manifesto genérico para o desenvolvimento de software

Nós, abaixo-assinados, temos uma opinião sobre o desenvolvimento de software. Estamos preocupados com o futuro de nossa profissão, e nossa experiência nos leva a chegar às seguintes conclusões:

- *Acreditamos em* um conjunto fixo de ideais imutáveis *em relação a* personalizar nossa abordagem para cada situação específica.
- *Acreditamos em* nos concentrar e discutir somente os assuntos que nos interessem *em relação ao* problema maior.
- *Acreditamos em* nossa opinião *em relação às* opiniões e à experiência de outras pessoas.
- *Acreditamos em* quaisquer obrigações que sejam "preto no branco" *em relação a* cenários do mundo real, com problemas complexos e soluções delicadas.
- *Acreditamos que* quando nossa abordagem for difícil de seguir, *então* isso mostrará somente o quanto ela é mais importante.
- *Acreditamos em* compor um conjunto arbitrário de mandamentos em relação a perceber que *nunca* é tão simples assim.
- *Acreditamos em* tentar estabelecer um movimento para promover a nossa visão *em relação a* algo que será genuinamente útil.
- *Acreditamos que* somos melhores desenvolvedores que aqueles que não concordem conosco *porque* eles não concordam conosco.

Ou seja, acreditamos que estamos fazendo o que é certo. E se você não acreditar, estará errado. E se você não fizer o que fazemos, estará fazendo

errado.

Tudo bem, tudo bem

Certo. Vou admitir. Estou brincando. Na maior parte.

Os manifestos

Talvez o manifesto mais famoso de desenvolvedores é *The Agile Manifesto* (*http://agilemanifesto.org/*), redigido em 2001 como um protesto contra os processos pesados e ineficientes que prejudicavam a entrega de softwares na(s) década(s) anterior(es). O mais recente *Craftsmanship Manifesto* (*http://manifesto.softwarecraftsmanship.org/*), infelizmente, é uma resposta para a degradação percebida em relação à importância das práticas técnicas e da responsabilidade por um bom código nos círculos Agile.

Há manifestos para outros movimentos de software, com destaque para o *manifesto GNU* (*http://www.gnu.org/gnu/manifesto.html*), *The Refactoring Manifesto* (*http://refactoringmanifesto.org/*) e *The Hacker Manifesto* (*http://en.wikipedia.org/wiki/Hacker_Manifesto*). E há outros.

Conheça os principais manifestos. Forme a sua própria opinião fundamentada sobre cada um deles.

> **PONTO-CHAVE** Conheça as metodologias de desenvolvimento, as tendências, os manifestos e os modismos.

É sério?

Bons desenvolvedores são apaixonados pelo seu trabalho. Eles se envolvem com o que fazem e procuram melhorar constantemente.

Isso é bom.

Quando um conjunto de práticas, ideais ou padrões que funcionem bem é identificado, é natural querer capturá-lo e compartilhá-lo com outros para o progresso da profissão. Atualmente, tornou-se popular registrar isso na forma de um *manifesto*. Como vimos, há vários deles.

Um manifesto, para a nossa arte, corresponde ao que os padrões de codificação são para o código: diretrizes úteis, ideais pelos quais lutar e orientação em direção às melhores práticas.

E assim como no caso dos padrões de codificação, guerras santas inúteis podem surgir em torno desses documentos. Alguns acólitos os veem como invariantes: obrigatórios, indelevelmente esculpidos em tabuletas preciosas de pedra por santos profetas. Eles rejeitam aqueles que não sigam o Único Caminho Verdadeiro.

Isso está longe de ser uma atitude produtiva.

> **PONTO-CHAVE** Subscreva-se aos manifestos de desenvolvimento que pareçam ser sensatos. Porém não os siga cegamente, e não os trate dogmaticamente.

Qualquer manifesto só pode ser uma afirmação geral de princípios, e *jamais* será o Único Caminho Verdadeiro. Por exemplo, os evangelizadores do Agile afirmam explicitamente que há várias maneiras de implementar os objetivos de seu manifesto; ele é somente uma tentativa de codificar as melhores práticas.

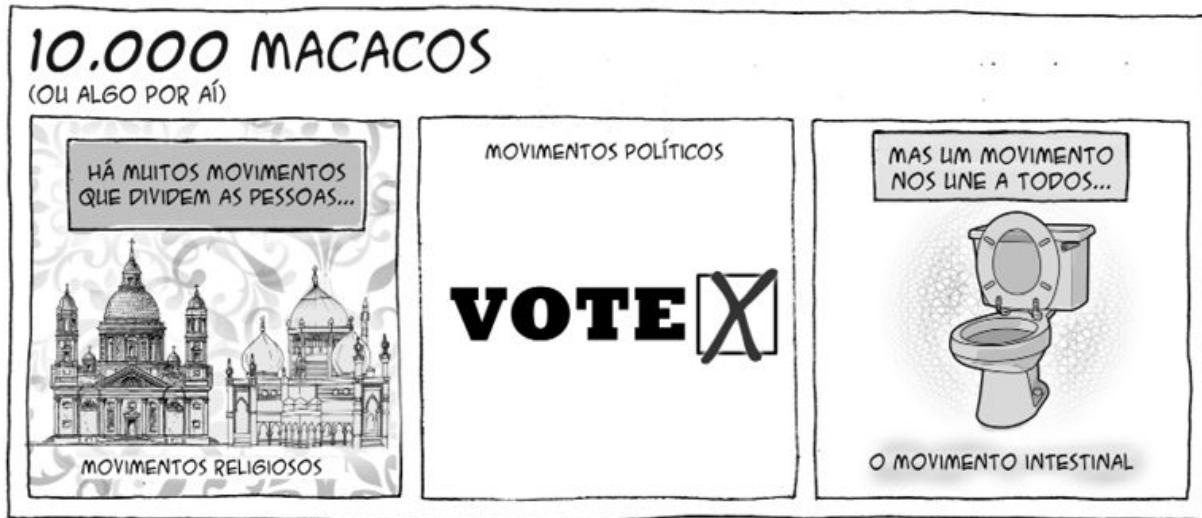
Se você se preocupa em se tornar um programador melhor, adote uma abordagem pragmática em relação a esse assunto. Aprenda com eles, compreenda as visões sobre o desenvolvimento que cada um defende. Utilize aquele que funcionar para você. Certifique-se de se manter atualizado; conheça as novas tendências e os catecismos em voga. Aprecie o que há de bom neles, mas não os siga cegamente. Avalie-os com a mente aberta.

A conclusão

Então os manifestos são tolos e sem sentido? Não. São úteis? Na maior parte. Ou seja, quando contêm boas informações e são usados de forma responsável de modo a incentivar conversas, e não quando usados como uma doutrina. Eles podem ser usados indevidamente? Sim! Facilmente. Mas o mesmo pode ocorrer com qualquer outra informação no mundo do desenvolvimento de software.

O que estaria em *seu* Manifesto para Desenvolvimento de Software? Aqui

está uma amostra do meu. Mas não o grave a ferro e fogo como *O Melhor Manifesto do Programador*. Ao menos, não até eu ter conseguido formar um movimento grande o suficiente para segui-lo.

- Importe-se com o código.
- Dê poder à sua equipe.
- Mantenha a simplicidade.
- Use o seu cérebro.
- Nada está gravado a ferro e fogo.
- Aprenda. Constantemente.
- Procure melhorar. Constantemente. (Você mesmo, sua equipe e o seu código.)
- Sempre gere valor considerando o longo prazo.

Perguntas

1. Quais "princípios" de desenvolvimento fundamentais você considera importantes?
2. Você concorda ou se aliena em relação a linhas de desenvolvimento como "Agile", "craftsmanship" e assim por diante? Até que ponto você concorda com cada um dos itens de seus manifestos?
3. O que você acha que esses manifestos têm a oferecer à comunidade de desenvolvimento?
4. Que tipo de dano eles *realmente* são capazes de provocar, se houver algum?
5. Ou você mantém a cabeça baixa e ignora esse tipo de assunto? Você *deveria* realmente seguir essas modas e modismos de software para manter o seu desenvolvimento pessoal?

Veja também

- *Viva para amar o aprendizado* (Capítulo 24) – Reserve tempo para conhecer as novas modas e os modismos do mercado.
- *Jogando segundo as regras* (Capítulo 15) – Escreva o seu próprio manifesto!

TENTE ISTO... Leia os manifestos listados anteriormente. Considere seus pontos de vista em relação a eles. Considere o que deveria estar em seu manifesto pessoal para o desenvolvimento de software.

10.000 MACACOS
(OU ALGO POR AÍ)

CAPÍTULO 38

Ode ao código

Toda poesia ruim tem origem em sentimentos genuínos.

– Oscar Wilde

Geraldo era um programador que trabalhava em uma equipe pequena.
O fato é que outros programadores criavam códigos que não valiam a pena.
A confusão era prejudicial, perturbadora, insana;
Eram detritos desumanos de uma mente profana.

Mas Geraldo tinha consciência. A situação não podia ficar assim.
Passava noites pensando em corrigir o que estava ruim
A estrutura interna horrível, os nomes confusos das variáveis,
E o fluxo de controle artificial, que eram consistentemente inviáveis.

Naquela época, a "regra dos escoteiros" era o que ele pretendia usar
Para os bugs corrigir e com o software inchado ao redor acabar.
Uma arrumação aqui, outra correção acolá, refatorar à esquerda e à direita.
Logo, logo, pensou ele, (com trabalho duro) farei a minha colheita.

Mas pobre Geraldo, com um plano em ação, ignorou um fato vital:
Para progredir, todos os programadores devem ter um compromisso real.
Seus colegas desleixados, desprezavam a regra para codificação
E continuaram a escrever despropósitos, enquanto ele tentava uma solução.

Um passo para a frente, dois para trás. Geraldo dançava.
Até aprender que de uma nova atitude militante ele precisava.
Equipes ágeis e um código limpo para dar uma melhorada.
Para isso, a equipe, e não o código, deverá ser abordada.

A lei de Conway descreve como da equipe o software é resultado
Um software agradável é produzido por um sistema bem lubrificado.
Se as engrenagens travam ou se quebram e não cumprem a obrigação.
Removê-las, pensou Geraldo, é a única solução.

E assim, com o padrão "Parametrizer de cima", começou a refatoração:
O gerente, a caminho de casa, recebeu um golpe com estupefação.
Ele caiu em um bueiro. O caso poderia ser chamado de assassinato.
Geraldo chamou de higiene. Um problema a menos havia, e isso parecia sensato.

Um a um, seus colegas, com fatos inusitados se depararam.
A equipe de QA, sem suspeitar, foi atingida por pratos que voaram.
(A lição aprendida com esse evento foi: jamais faça uma reunião
em um jantar com louças ruins e tendências à assombração.)

Os programadores que causaram tal ira um fim sangrento tiveram.
A "impressora pegou a gravata de um"; sua cara, consertar não puderam.
Outro, a caminho de uma pausa, na escada tropeçou,
E uma pilha de manuais de Unix, mortalmente em sua direção voou.

A vida de Geraldo melhorou bastante; na equipe somente havia
um programador, um administrador de sistemas e o sujeito que a porta abria.
O problema com essa equipe, Geraldo logo descobriria:
O código não piorou – bom! –, mas mal havia mudanças, pois programador não havia.

O progresso era duro e lento, embora Geraldo tentasse heroicamente.
Os prazos eram ameaçadores e sempre se esgotavam rapidamente.
Com funcionalidades infelizmente ausentes, o projeto não tinha razão.
Então, um dia, um guarda apareceu e pôs Geraldo na prisão.

A moral dessa simples história é pensar bem antes de reagir
Quando programadores insensíveis no desespero o fizerem cair.
A única maneira sensata com que você deve responder

É agir como um britânico e um nível controlado de fúria manter.

Codificação é um problema de pessoas

Espero que você tenha lido o capítulo sobre ética, portanto é provável que você concorde que não é aconselhável realizar um corte dramático dos membros de sua equipe de software com baixo desempenho. Entretanto como você *deve* reagir ao trabalhar com membros da equipe que não tenham um desempenho adequado ou que pareçam piorar deliberadamente o código?

E se os líderes da equipe de software não perceberem nem compreenderem o problema? E se, que os céus não permitam, eles forem parte do próprio problema?

Infelizmente, considerando a parte sofredora do código, isso não é totalmente incomum. Embora algumas equipes estejam repletas de programadores maravilhosos, muitas não estão. A menos que você seja anormalmente abençoado em sua carreira de codificação, em algum ponto, você se verá em situações complicadas que parecem não ter solução.

> **PONTO-CHAVE** Com frequência, a parte complicada do desenvolvimento de software não está nos aspectos técnicos do código; está relacionada às pessoas.

Quando os programadores simplesmente parecem não entender, e falham em entender que estão piorando a situação em vez de melhorá-la, você deve responder.

Considere a introdução de práticas que promovam a responsabilidade pelo código e que mostrem (de uma maneira que evite a atribuição de culpa) como trabalhar da maneira mais eficiente possível: introduza programação aos pares, programas de orientação, reuniões de revisão de design ou outras práticas desse tipo.

Faça de você mesmo um excelente exemplo. Não caia na armadilha desses hábitos ruins; é muito fácil perder o entusiasmo e tomar atalhos porque todos os demais o fazem. Se não puder derrotá-los, *não* junte-se a eles.

PONTO-CHAVE Quando você estiver cercado por programadores que não se importem com o código, mantenha atitudes saudáveis. Tome cuidado para não absorver práticas ruins por osmose.

Não será simples nem rápido mudar uma cultura de codificação e levar o desenvolvimento de volta aos princípios saudáveis. Mas isso não significa que não possa ser feito.

Perguntas

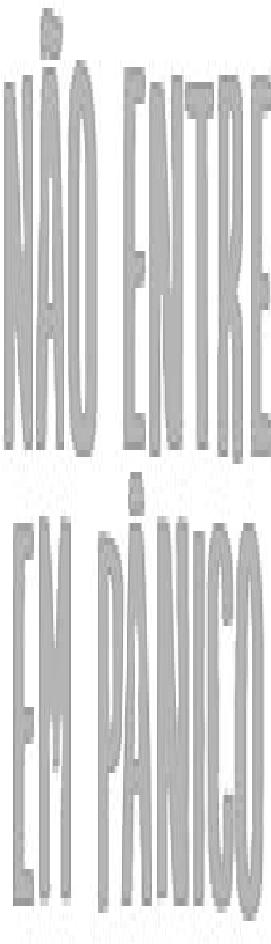
1. Quão saudável é a sua equipe de desenvolvimento atual?
2. Como você pode reconhecer rapidamente quando um desenvolvedor não está tendo um desempenho tão bom quanto deveria?
3. O que é mais provável: pessoas trabalhando desleixadamente de propósito ou elas serem desleixadas porque não conseguem ver como poderiam trabalhar melhor?
4. Como você pode ter certeza de que você mesmo não está adotando práticas desleixadas? Como é possível evitar que você se deixe levar pelas práticas ruins no futuro?

Veja também

- *Importar-se com o código* (Capítulo 1) – Você *deve* se importar com o código. Mas seria possível você se importar *demais*?
- *O programador ético* (Capítulo 28) – Por favor, releia este capítulo, somente para o caso de você estar prestes a se deixar levar por uma fúria assassina.
- *Chafurdando na lama* (Capítulo 7) – Como lidar com a confusão deixada pelos colegas que não sabem fazer melhor.

TENTE ISTO... Considere se você adotou algum hábito ruim recentemente. Como isso pode ser corrigido?

10.000 MACACOS
(OU ALGO POR AÍ)

TENTATIVA DE CRIAR CÓDIGO

Epílogo

Nem todo fim é o objetivo. O fim de uma melodia não é o seu objetivo; apesar disso, se uma melodia não atingir o seu fim, ela não terá alcançado o seu objetivo.

– Friedrich Nietzsche

Bem distante, nas profundezas inexploradas do limite inimaginável do braço espiral ocidental da Galáxia, encontra-se um pequeno sol amarelo sem importância. Orbitando em torno dele, a uma distância de aproximadamente 148 milhões de quilômetros, está um pequeno planeta verde-azulado extremamente insignificante, cujas formas de vida descendentes dos macacos são tão incrivelmente primitivas que elas ainda acham que os programas de computador sejam uma ótima ideia.

Esse planeta tem – ou melhor, tinha – um problema a saber: a maioria dos programadores escrevia códigos bem pobres, praticamente o tempo todo, mesmo que estivesse sendo paga para fazer um bom trabalho. Muitas soluções foram sugeridas para esse problema, porém a maior parte delas dizia respeito principalmente à educação dos programadores, o que é estranho porque, em geral, os programadores não queriam ser educados.

E o problema continuou; muitos do códigos produzidos eram ruins e a maioria dos usuários se sentia miserável, até mesmo aqueles que podiam escrever bons programas de computador.[1]

Muito bem: você chegou ao fim do livro. Foram muitos capítulos digeridos. (Se você simplesmente pulou para cá para arruinar o final: *o mordomo é o culpado*. Agora volte e leia por quê.)

Nas últimas centenas de páginas, você conheceu técnicas para escrever códigos tecnicamente elegantes, para criar designs maravilhosos e para

construir sistemas pragmáticos, possíveis de manter. Conheceu abordagens para lidar com código legado e viu como trabalhar de modo eficiente com outras pessoas.

Porém todo esse conhecimento adquirido e a compreensão de habilidades específicas não o ajudará nem um pouco se você não tiver a *atitude* correta: o desejo e a paixão por um bom trabalho.

Você tem isso?

Atitude

A atitude é o que separa os *bons* programadores dos *ruins*; é o que distingue os programadores *excepcionais* dos meramente *adequados*.

A atitude supera a habilidade técnica: um conhecimento complexo de uma linguagem de programação não garante que um código possa ser mantido. Entender muitos modelos de programação nem sempre resulta em designs elegantes. É a sua atitude que determina se o seu código será "bom" e se será um prazer trabalhar com você.

A definição no dicionário de *atitude* é:

> **Atitude:** (s.f.) *a-ti-tu-de*
>
> 1. Um estado de espírito ou um sentimento; uma disposição.
> 2. A posição de uma aeronave em relação a um ponto de referência.

A primeira definição não é surpreendente, porém a segunda, na realidade, é mais reveladora que a primeira.

Há três linhas de eixo imaginárias que passam por uma aeronave; uma de asa a asa, uma do nariz até a cauda e outra que passa verticalmente no ponto em que as outras duas linhas se cruzam. Um piloto posiciona a sua aeronave em torno desses eixos; estes definem o ângulo de aproximação da aeronave. Isso é conhecido como a *atitude* da aeronave. Se você aplicar um pouco de potência na aeronave quando ela tiver a atitude incorreta, ela acabará errando totalmente o alvo. Um piloto deve monitorar constantemente a atitude de sua aeronave, especialmente em momentos críticos como na decolagem e no pouso.

Podemos fazer um paralelo com nossas experiências em desenvolvimento de software. A atitude do avião define o seu ângulo de aproximação, e *nossa* atitude define o nosso ângulo de abordagem ao código. Independentemente do quão tecnicamente competente você seja como programador, se suas habilidades não forem reforçadas com atitudes saudáveis, o seu trabalho irá sofrer.

Com a atitude incorreta, você poderá seguir quilômetros na direção errada. A atitude pode representar o sucesso ou o fracasso de um projeto de software, portanto é fundamental manter o ângulo correto de aproximação para a programação. Sua atitude irá atrapalhar ou promover o seu crescimento pessoal. Para sermos programadores melhores, devemos garantir que adotaremos as atitudes corretas.

PONTO-CHAVE Sua atitude determina como você irá progredir como programador. Procure adotar atitudes melhores para se tornar um programador melhor.

Vá em frente e codifique

Importe-se com um bom código e procure criá-lo das melhores maneiras possíveis. Aprenda sempre. Aprenda a fazer design, a codificar e a cooperar. Procure sempre trabalhar com excelentes engenheiros que o desafiem e o incentivem a melhorar. Seja diligente, consciencioso e profissional.

Aprecie a programação. E, acima de tudo, aprecie o fato de melhorar!

TENTE ISTO... Leia este livro novamente daqui a alguns meses. Revise o material e veja o que chamará a sua atenção da próxima vez. Tente responder de novo às perguntas e observe como suas perspectivas, a experiência e a sua compreensão mudaram. Se você for diligente e se concentrar em praticar as técnicas deliberadamente, você se surpreenderá com o seu progresso.

1 Com desculpas ao falecido e ótimo Douglas Adams.

Sobre o autor

Pete Goodliffe é programador, colunista que escreve sobre desenvolvimento de software, músico e autor. Ele jamais permanece no mesmo ponto da cadeia alimentar de software; seus projetos variam de implementação de sistemas operacionais a codecs de áudio, aplicações multimídia ou, ainda, firmware embarcado, desenvolvimento para iOS e aplicações desktop. Tem paixão por *curry* e não usa sapatos.

O livro popular de desenvolvimento de Pete – *Code Craft* – é uma investigação prática e interessante de tudo o que se busca com a programação. Em aproximadamente 600 páginas. Não é brincadeira! O livro já foi traduzido para diversos idiomas. Pete é autor de uma coluna chamada "Becoming a Better Programmer" (*Como ser um programador melhor*) para uma revista e contribuiu com vários livros sobre desenvolvimento de software, além de dar palestras regularmente sobre esse assunto.

Colofão

O animal na capa de *Como ser um programador melhor* é o peixe-voador de duas asas (*Exocoetidae*). O peixe-voador pode ser distinguido pelas suas nadadeiras peitorais anormalmente grandes. Como o nome sugere, o peixe dá saltos eficazes, impulsionando a si mesmo para fora da água em direção ao ar, e move sua cauda aproximadamente 70 vezes por segundo. No início do século 21, os peixes-voadores eram estudados como modelos para o desenvolvimento de aviões. Depois que está no ar, o peixe abre suas nadadeiras semelhantes a asas e as aponta para cima, o que lhe possibilita percorrer distâncias consideráveis. A nadadeira peitoral – ou “asa” – tem um formato aerodinâmico semelhante à asa de um pássaro. No final do voo, o peixe retorna ao mar dobrando as nadadeiras peitorais ou deixa a cauda cair na água a fim de se erguer para outro voo. O recorde é de 45 segundos de voo, conforme registrado por uma equipe de TV japonesa em 2008.

Sua capacidade de voar geralmente é usada como mecanismo de defesa contra os predadores, que incluem golfinhos, atum, marlim, pássaros, lulas e toninhas. São comercialmente pescados com redes verticais no Japão, no Vietnã e na China e com puçás na Indonésia e na Índia. Nas Ilhas Salomão, os peixes-voadores são capturados durante o voo em redes fixas em canoas. A pesca comercial dessa espécie é feita em completa escuridão, quando não há luar, pois os peixes são atraídos para tochas acesas. O peixe-voador se alimenta basicamente de plâncton e vive em todos os oceanos, principalmente em águas mais mornas, tropicais ou subtropicais.

Muitos dos animais nas capas dos livros da O’Reilly estão ameaçados; todos eles são importantes para o mundo. Para saber como você pode ajudar, acesse *animals.oreilly.com*.

Programação
em Baixo Nível
C, Assembly e execução de programas
na arquitetura Intel 64
Igor Zhirkov
novatec
Apress

Programação em Baixo Nível

Zhirkov, Igor

9788575226704

576 páginas

Compre agora e leia

Conheça a linguagem Assembly e a arquitetura do Intel 64, torne-se proficiente em C e entenda como os programas são compilados e executados até o nível das instruções de máquina, permitindo-lhe escrever um código robusto e de alto desempenho. Programação em baixo nível explica a arquitetura do Intel 64 como resultado da evolução da arquitetura de von Neumann. O livro o ensina a usar a versão mais recente da linguagem C (C11) e a linguagem Assembly desde o básico. Todo o caminho, do código-fonte à execução do programa, incluindo a geração de arquivos-objeto ELF, além das ligações estática e dinâmica, será discutido. Há exemplos de código e exercícios, junto com as melhores práticas de programação. Os recursos de otimização e os limites dos compiladores modernos serão analisados, permitindo-lhe promover um equilíbrio entre a legibilidade do programa e o desempenho. O uso de diversas técnicas para ganho de desempenho, por exemplo,

instruções SSE e pre-fetching, será demonstrado. Assuntos relevantes em ciência da computação, como os modelos de computação e as gramáticas formais, também serão tratados, explicando-se sua importância prática. Programação em baixo nível ensina os programadores a: escrever livremente em linguagem Assembly; compreender o modelo de programação do Intel 64; escrever um código robusto e fácil de manter em C11; acompanhar o processo de compilação e decifrar as listagens em Assembly; depurar erros em código Assembly compilado; usar modelos de computação apropriados para reduzir drasticamente a complexidade dos programas; escrever códigos críticos quanto ao desempenho; compreender o impacto de um modelo de memória fraco em aplicações com várias threads.

Compre agora e leia

O'REILLY
Padrões para Kubernetes
Elementos reutilizáveis no design de aplicações nativas de nuvem
Bilgin Ibryam
Roland Huß
novatec

seguintes classes de padrões: Padrões básicos, que incluem princípios e práticas essenciais para desenvolver aplicações nativas de nuvem com base em contêineres. Padrões comportamentais, que exploram conceitos mais específicos para administrar contêineres e interações com a plataforma. Padrões estruturais, que ajudam você a organizar contêineres em um Pod para tratar casos de uso específicos. Padrões de configuração, que oferecem insights sobre como tratar as configurações das aplicações no Kubernetes. Padrões avançados, que incluem assuntos mais complexos, como operadores e escalabilidade automática (autoscaling).

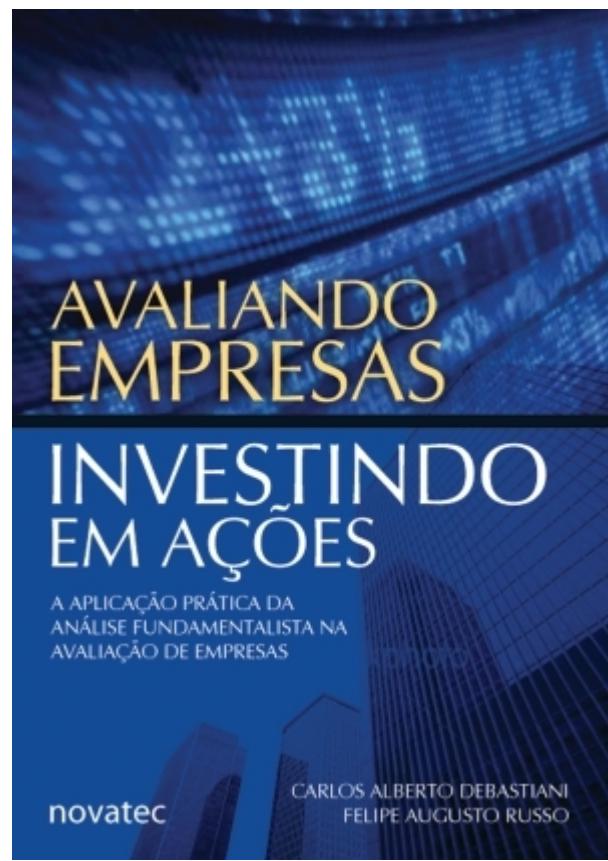
[Compre agora e leia]

CANDLESTICK
Um método para ampliar lucros na Bolsa de Valores
novatec
Carlos Alberto Debastiani

Candlestick

Debastiani, Carlos Alberto

9788575225943

200 páginas

Compre agora e leia

A análise dos gráficos de Candlestick é uma técnica amplamente utilizada pelos operadores de bolsas de valores no mundo inteiro. De origem japonesa, este refinado método avalia o comportamento do mercado, sendo muito eficaz na previsão de mudanças em tendências, o que permite desvendar fatores psicológicos por trás dos gráficos, incrementando a lucratividade dos investimentos. Candlestick – Um método para ampliar lucros na Bolsa de Valores é uma obra bem estruturada e totalmente ilustrada. A preocupação do autor em utilizar uma linguagem clara e acessível a torna leve e de fácil assimilação, mesmo para leigos. Cada padrão de análise abordado possui um modelo com sua figura clássica, facilitando a identificação. Depois das características, das peculiaridades e dos fatores psicológicos do padrão, é apresentado o gráfico de um caso real aplicado a uma ação negociada na Bovespa. Este livro

possui, ainda, um índice resumido dos padrões para pesquisa rápida na utilização cotidiana.

Compre agora e leia

AVALIANDO
EMPRESAS
INVESTINDO
EM AÇÕES
A APLICAÇÃO PRÁTICA DA
ANÁLISE FUNDAMENTALISTA NA
AVALIAÇÃO DE EMPRESAS
CARLOS ALBERTO DEBASTIANI
FELIPE AUGUSTO RUSSO
novatec

Marcos Abe
MANUAL DE
ANÁLISE
TÉCNICA
ESSÊNCIA E ESTRATÉGIAS AVANÇADAS
TUDO O QUE UM INVESTIDOR PRECISA SABER PARA
PROSPERAR NA BOLSA DE VALORES ATÉ EM TEMPOS DE CRISE
novatec

meio do gerenciamento de capital. Destina-se a iniciantes na bolsa de valores e investidores que ainda não desenvolveram uma metodologia própria para operar lucrativamente.

Compre agora e leia